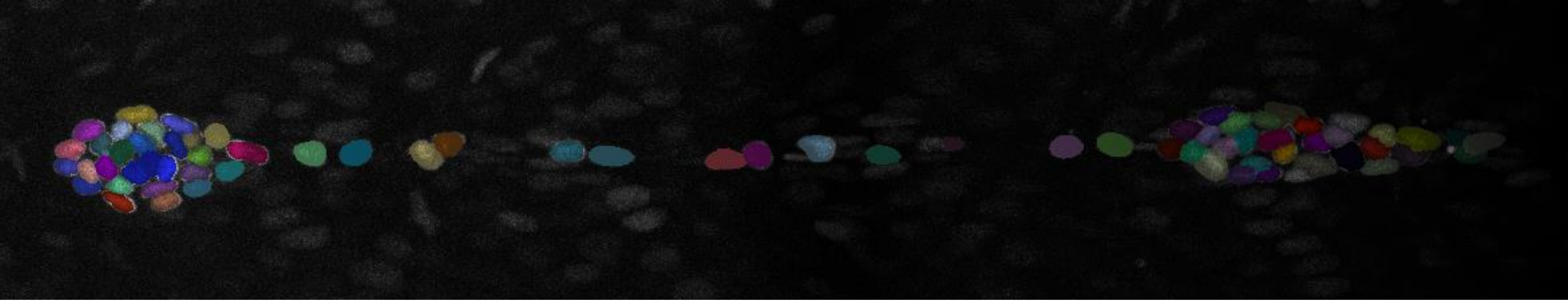


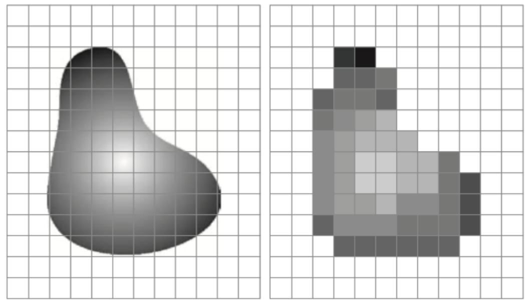
Digital Image Processing

Lecture 03: Deep Learning Crash Course (20th April, 2023)

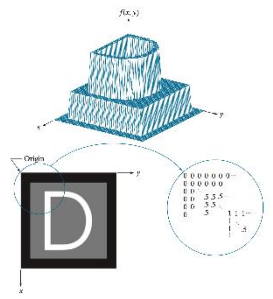
Johannes Stegmaier
Institute of Imaging and Computer Vision
RWTH Aachen University

`http://www.lfb.rwth-aachen.de`
`johannes.stegmaier@lfb.rwth-aachen.de`

Review: Digital Imaging Fundamentals



Sampling and Quantization



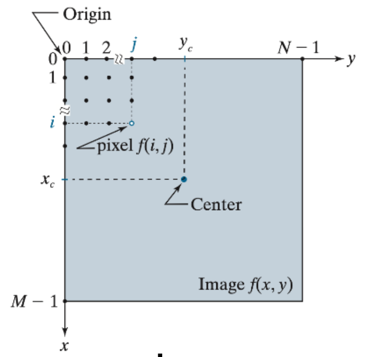
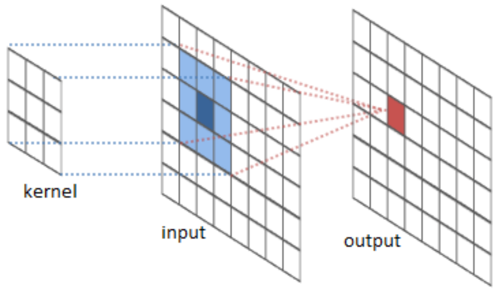
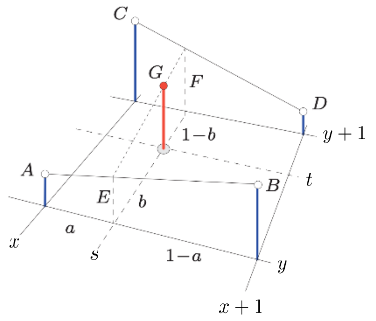
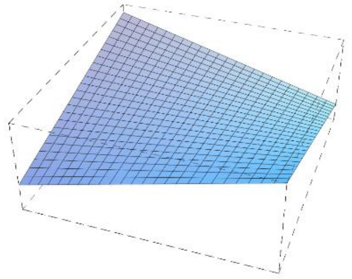


Image Representations



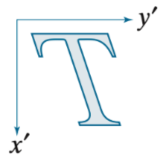
Basic Operations on Images





Interpolation Methods for 2D Images

$$\begin{bmatrix}x'\\y'\\1\end{bmatrix} = \mathbf{A}\begin{bmatrix}x\\y\\1\end{bmatrix} = \begin{bmatrix}a_{11}&a_{12}&a_{13}\\a_{21}&a_{22}&a_{23}\\0&0&1\end{bmatrix}\begin{bmatrix}x\\y\\1\end{bmatrix}$$



Geometric Image Operations

Agenda for Today's Lecture

- Artificial Neural Networks
- Training Multilayer Perceptrons
- Convolutional Neural Networks
- Summary, Preview and Literature

Motivation

- Neural networks are **loosely inspired by neuroscience**.
- **Artificial neurons** are the building blocks of multilayered architectures.
- Analogous to biological neurons, artificial neurons receive inputs from other neurons, combine the inputs in a certain way and compute an activation value as their output.
- Activation functions are used to incorporate nonlinearity to the processing.
- The goal is NOT to perfectly model the brain, but to have general function approximation machines.
- Neural networks excel at learning a representation of the training data to perform predictions on unseen samples.

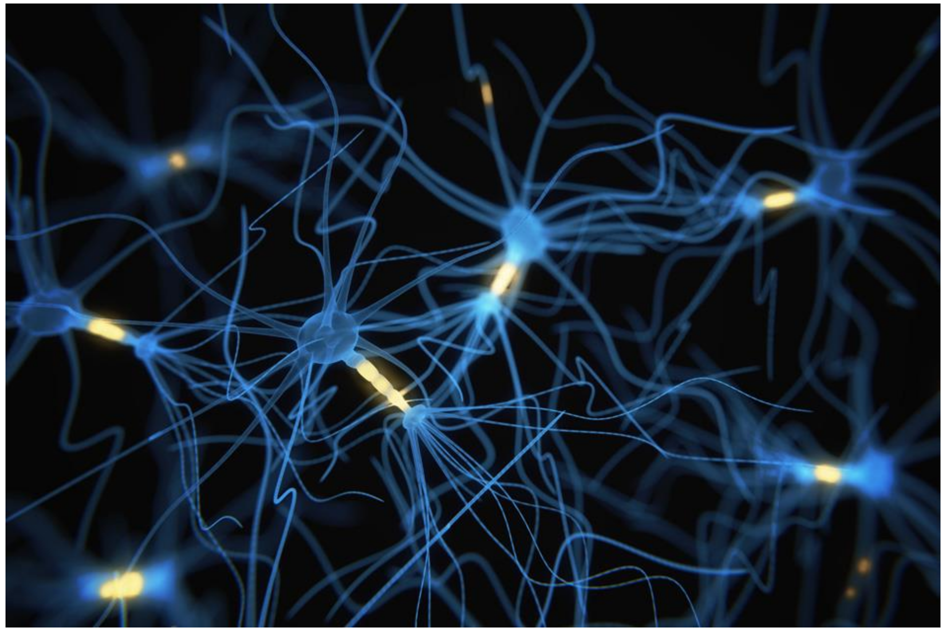


Image: http://news.mit.edu/sites/mit.edu.newsoffice/files/images/2017/MIT-Label-Neurons_0.jpg

Basic Components of Feedforward Neural Networks

- The perceptron (Rosenblatt, 1958) was introduced to model a linear combination of multiple inputs and using a discontinuous activation function to set the output to 0 or 1.

$$y(\mathbf{x}, \mathbf{w}) = f_{\text{step}}(w_0 + \mathbf{w}^T \mathbf{x})$$

- The term $w_0 + \mathbf{w}^T \mathbf{x}$ describes a plane in the D-dimensional feature space, with w_0 being the **bias weight** (offset of the plane from the origin).

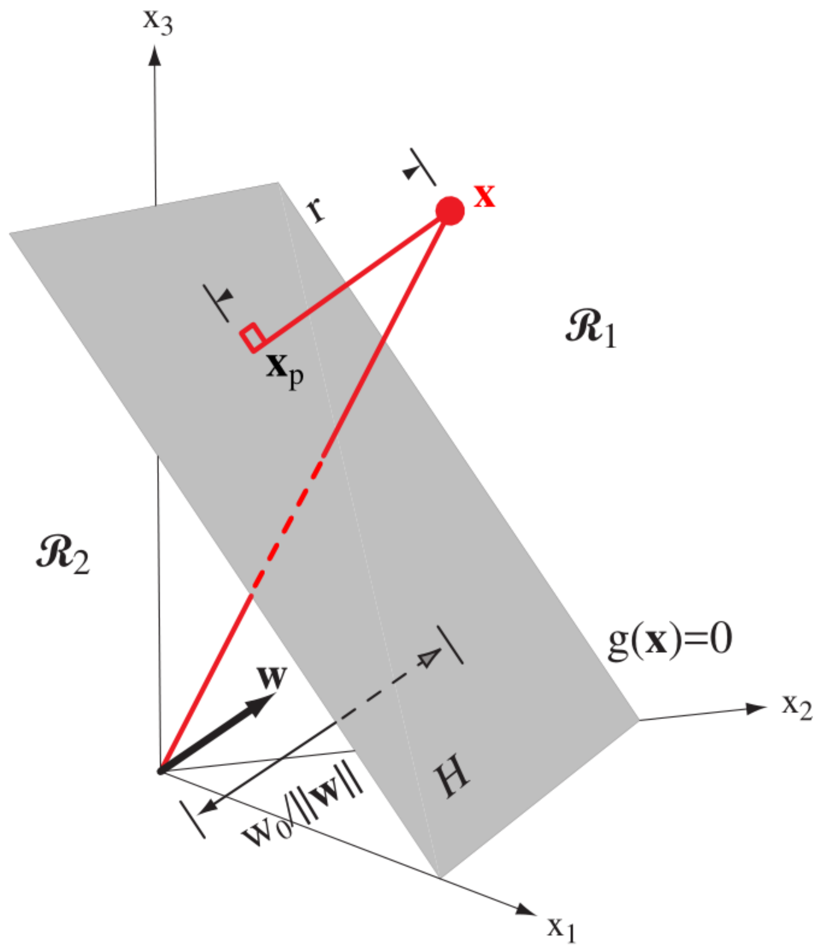


Image: R. Duda, P. Hart, D. Stork: Pattern Classification, John Wiley & Sons, 2001.

Basic Components of Feedforward Neural Networks

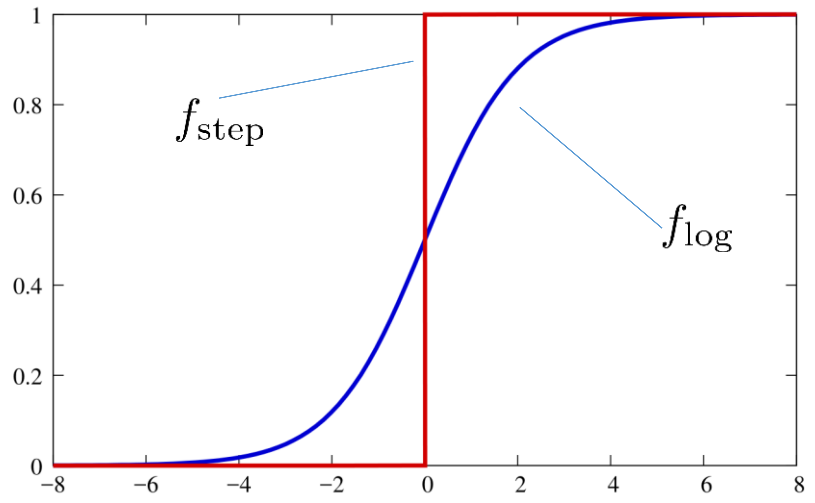
- The perceptron (Rosenblatt, 1958) was introduced to model a linear combination of multiple inputs and using a discontinuous activation function to set the output to 0 or 1.

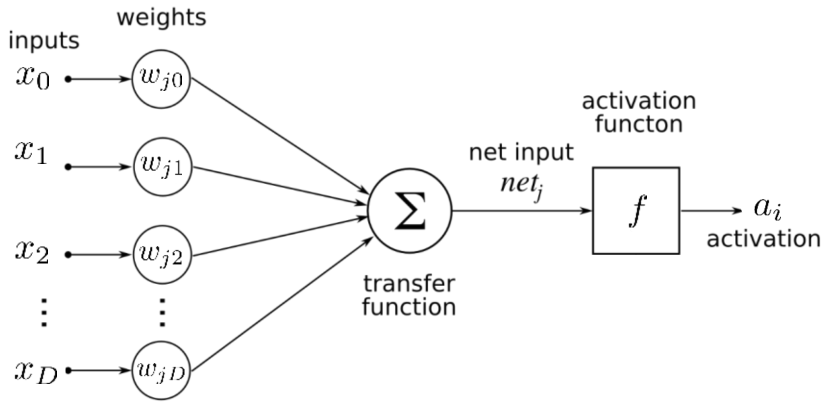
$$y(\mathbf{x}, \mathbf{w}) = f_{\text{step}}(w_0 + \mathbf{w}^T \mathbf{x})$$

- The term $w_0 + \mathbf{w}^T \mathbf{x}$ describes a plane in the D-dimensional feature space, with w_0 being the **bias weight** (offset of the plane from the origin).

- The activation function determines the output of the perceptron:

$$f_{\text{step}}(z) = \begin{cases} 1, \text{if } z > 0 \\ 0, \text{otherwise.} \end{cases}$$





Components of an artificial neuron / perceptron.

Image: R. Duda, P. Hart, D. Stork: Pattern Classification, John Wiley & Sons, 2001.

Basic Components of Feedforward Neural Networks

- The perceptron (Rosenblatt, 1958) was introduced to model a linear combination of multiple inputs and using a discontinuous activation function to set the output to 0 or 1.

$$y(\mathbf{x}, \mathbf{w}) = f_{\text{step}}(w_0 + \mathbf{w}^T \mathbf{x})$$

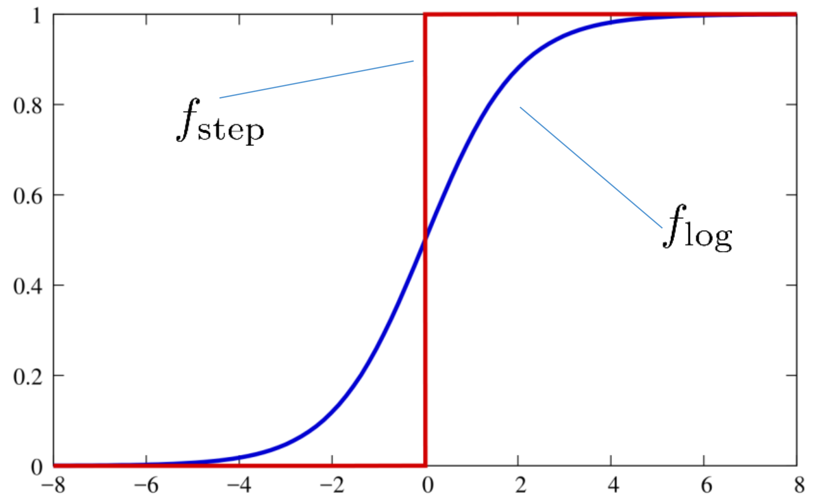
- Neurons in multilayer perceptrons use a smoothed variant of this, to be able to compute gradients:

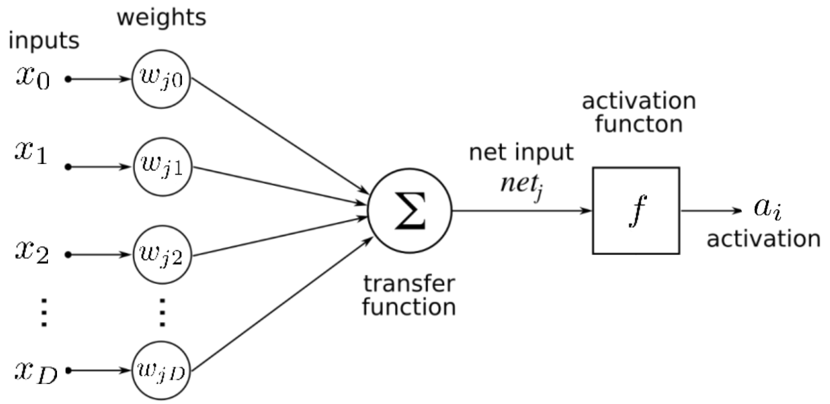
$$y(\mathbf{x}, \mathbf{w}) = f_{\log}(w_0 + \mathbf{w}^T \mathbf{x})$$

- For instance, using the logistic sigmoidal function instead of the step function:

$$f_{\log}(z) = \frac{1}{1 + e^{-z}}$$

- Properties: monotonically increasing, continuous and differentiable, $\lim_{z\to\infty} = 1$, $\lim_{z\to-\infty} = 0$, $f_{\log}(z) = 1 - f_{\log}(-z)$

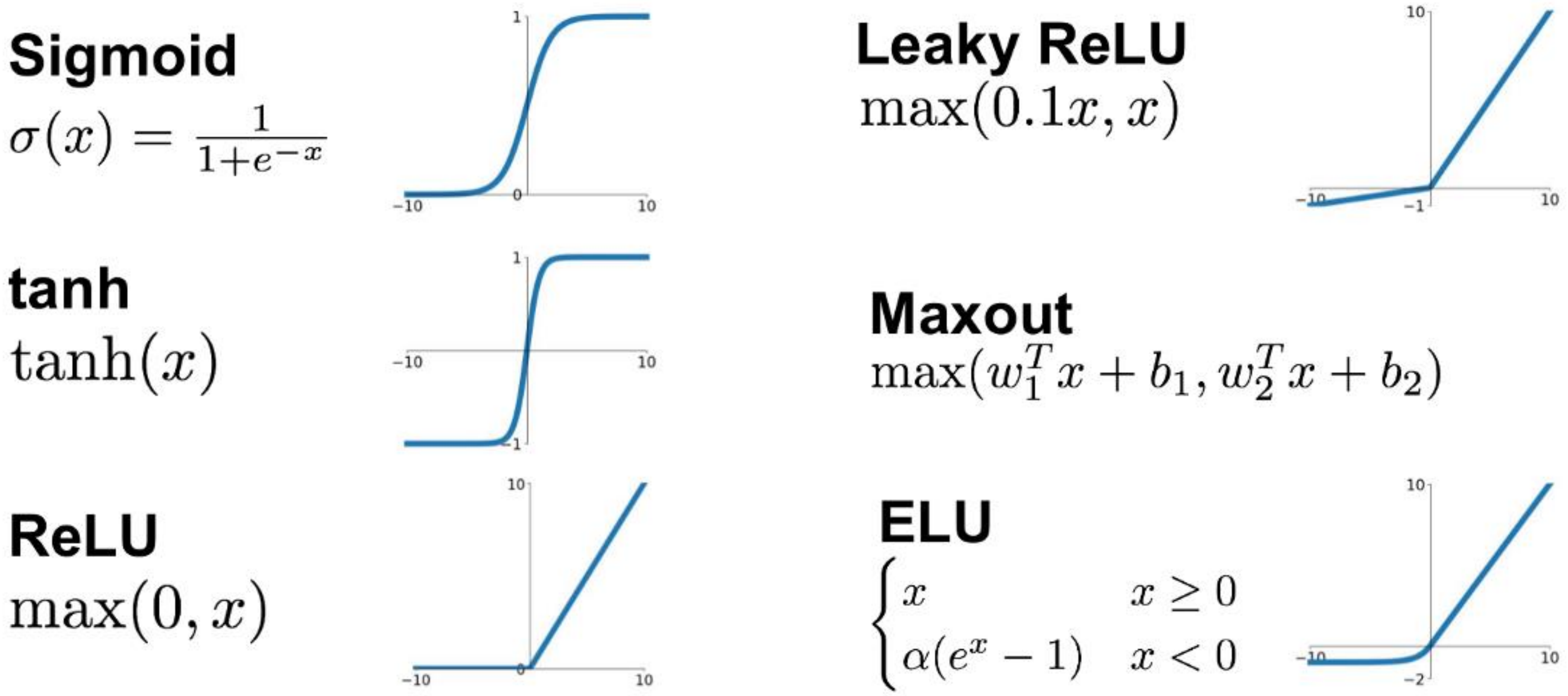




Components of an artificial neuron / perceptron.

Image: R. Duda, P. Hart, D. Stork: Pattern Classification, John Wiley & Sons, 2001.

Other Examples of Nonlinear Activation Functions



Sigmoid

$\sigma(x) = \frac{1}{1+e^{-x}}$

tanh

$\tanh(x)$

ReLU

$\max(0, x)$

Leaky ReLU

$\max(0.1x, x)$

Maxout

$\max(w_1^T x + b_1, w_2^T x + b_2)$

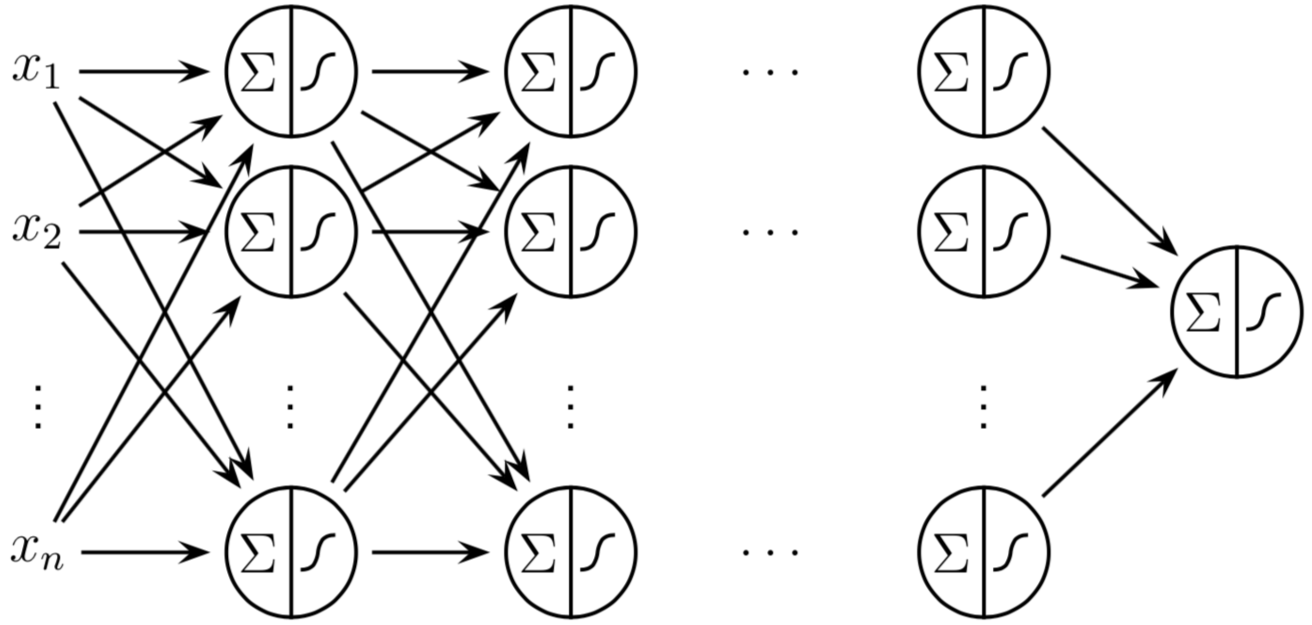
ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$

Image: https://qph.fs.quoracdn.net/main-qimg-65a7c3bf75549bad04875d0e789bb5bf

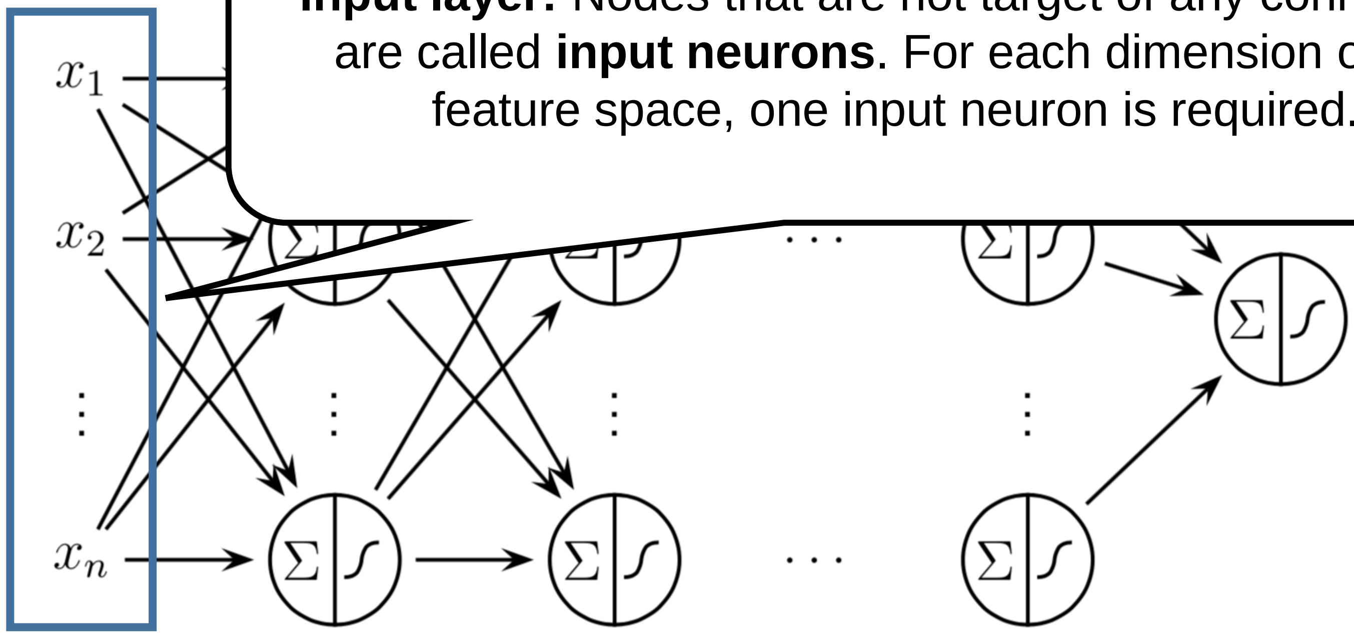
Multilayer Perceptron Architecture

- A **multilayer perceptron (MLP)** is a finite acyclic graph.
- The nodes are neurons that perform a linear combination of their inputs and transform the sum using a nonlinear activation function.
- Neurons of the i-th layer serve as input features for layer $i+1$.
- By combining many neurons, very complex functions can be approximated.
- Due to being an acyclic graph, the neurons can be organized in layers (input layer, hidden layer(s), output layer).



Adapted from M. Riedmiller, Lecture “Machine Learning”, University of Freiburg, 2011.

Multilayer Perceptron Architecture

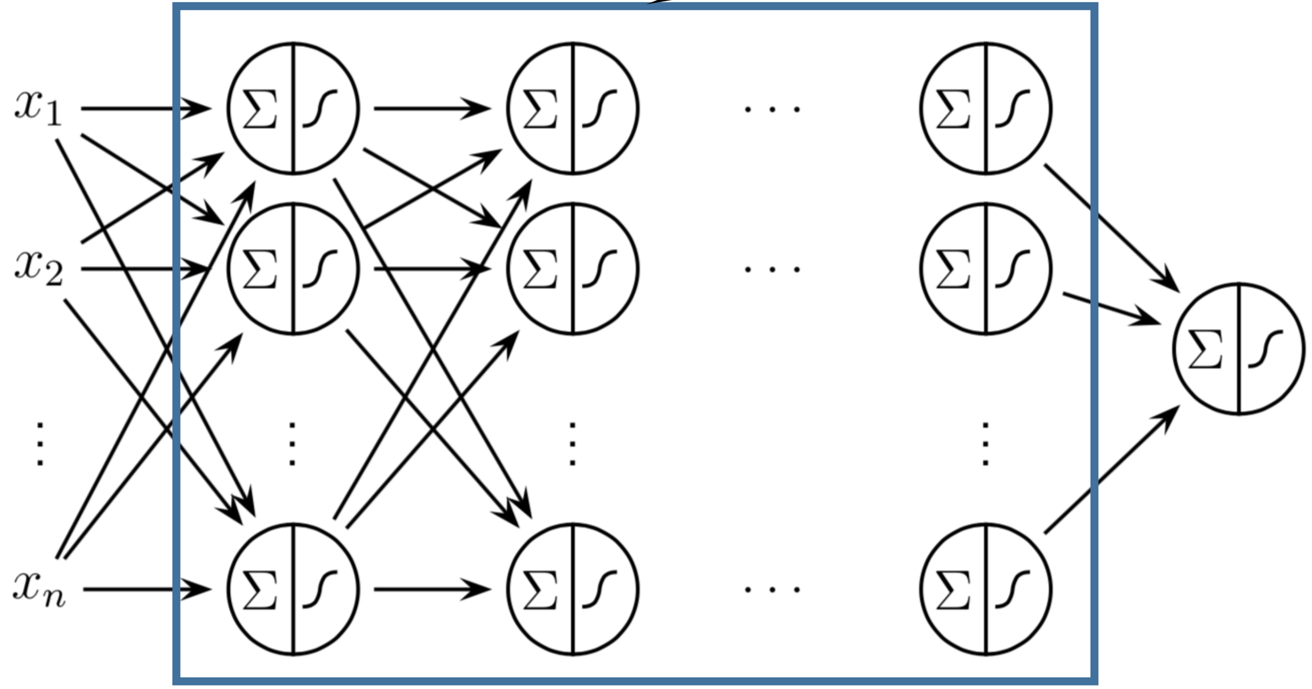


Input layer: Nodes that are not target of any connection are called **input neurons**. For each dimension of the feature space, one input neuron is required.

Adapted from M. Riedmiller, Lecture “Machine Learning”, University of Freiburg, 2011.

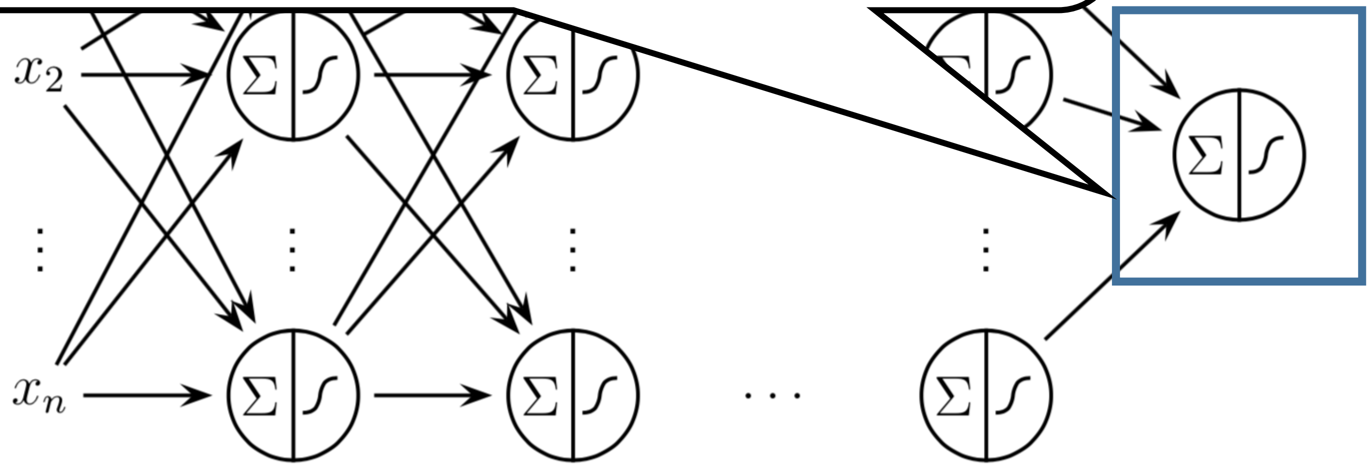
Multilayer Perceptron Architecture

Hidden layer(s): Nodes that are neither input nor output neurons are called **hidden neurons**.



Adapted from M. Riedmiller, Lecture “Machine Learning”, University of Freiburg, 2011.

Multilayer Perceptron Architecture

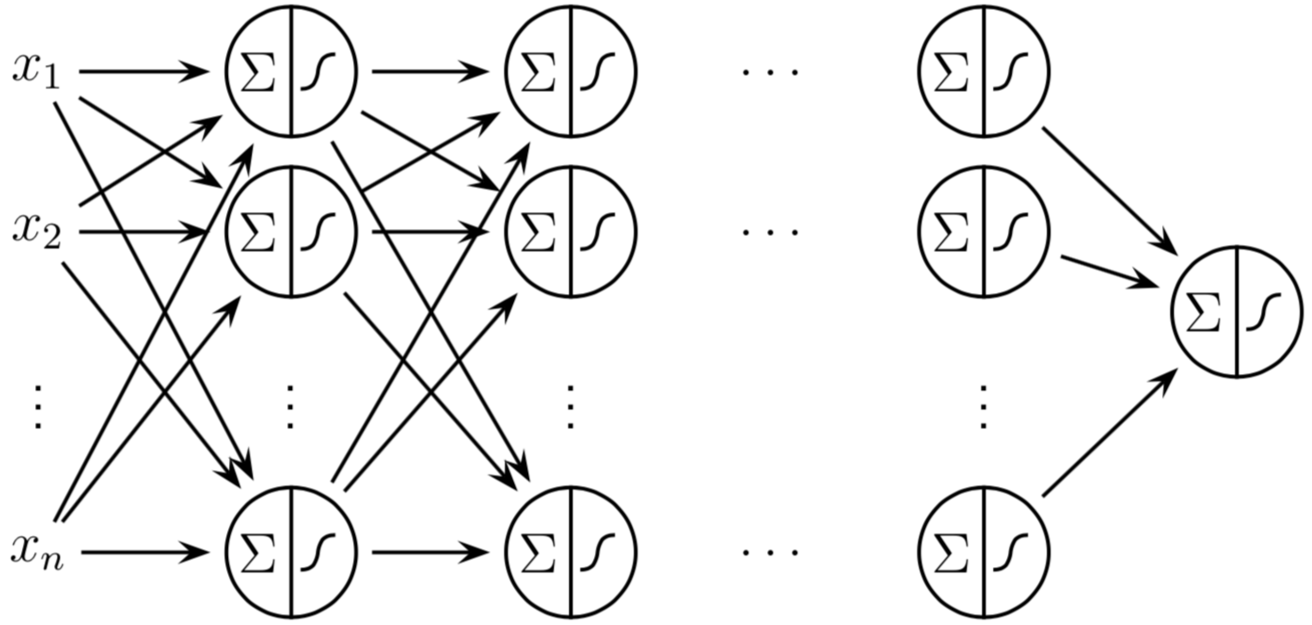


Output layer: Nodes that are no source of any connection are called **output neurons**. The number of output neurons depends on the learning task (target values of the training data).

Adapted from M. Riedmiller, Lecture “Machine Learning”, University of Freiburg, 2011.

Multilayer Perceptron Notation

- Connections that skip one or more layers are called **shortcut** or **skip-connections**.
- MLPs are mostly fully-connected, *i.e.*, all neurons of one layer connect to all neurons of the next layer.
- Notation convention:
  - $\mathrm{Succ}(i)$ is the set of neurons j that have a connection from $i \rightarrow j$.
  - $\mathrm{Pred}(i)$ is the set of neurons j that have a connection from $j \rightarrow i$.
- Connections are weighted with a real number. For instance, w_{ji} denotes the weight of connection $i \rightarrow j$.
- All hidden neurons have a **bias weight**. A hidden neuron i has a bias weight named w_{i0}.



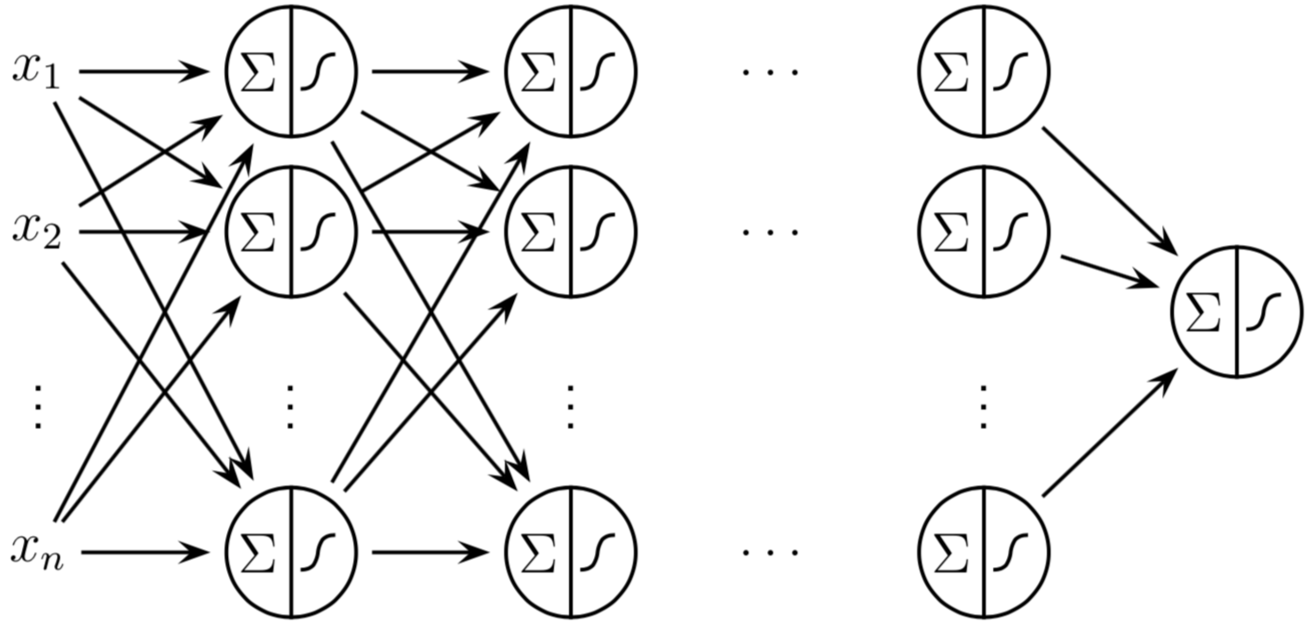
Adapted from M. Riedmiller, Lecture "Machine Learning", University of Freiburg, 2011.

Multilayer Perceptron Notation

- Hidden and output neurons receive input from their predecessors which is abbreviated net_i .
- All neurons have an activation/output denoted by a_i.
- To apply a pattern $\mathbf{x} = [x_1, \dots, x_D]^T$ to the MLP:
  - Present i-th feature to the i-th input neuron $a_i \leftarrow x_i$
  - Calculate net_i and a_i of all hidden and output neurons as soon as the activations a_j for all predecessors $j \in \mathrm{Pred}(i)$ have been computed:

$$net_i \leftarrow w_{i0} + \sum_{j \in \mathrm{Pred}(i)} w_{ij} a_j$$
$$a_i \leftarrow f_{\mathrm{sig}}(net_i)$$

  - The network output is given by the activations a_i of the output neurons.



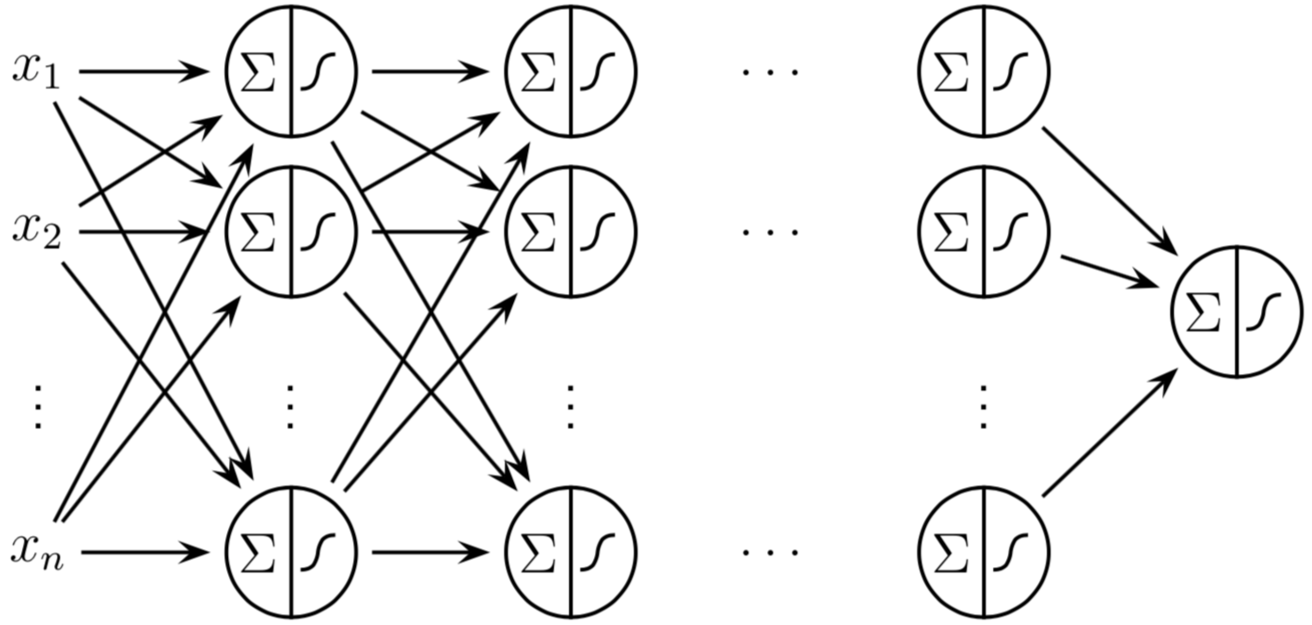
Adapted from M. Riedmiller, Lecture “Machine Learning”, University of Freiburg, 2011.

Pseudocode for the Forward Pass Algorithm

Input: Feature vector $\mathbf{x}$, MLP and topological order of the neurons.

Output: Result of passing $\mathbf{x}$ through the MLP.

```
1.  for all input neurons $i$ do
2.      set $a_i \leftarrow x_i$
3.  end for
4.  for all hidden and output neurons $i$ in
    topological order do
5.      set $net_i \leftarrow w_{i0} + \sum_{j \in \text{Pred}(i)} w_{ij} a_j$
6.      set $a_i \leftarrow f_{\text{sig}}(net_i)$
7.  end for
8.  for all output neurons $i$ do
9.      assemble $a_i$ in output vector $\mathbf{y}$
10. end for
11. return $\mathbf{y}$
```



Adapted from M. Riedmiller, Lecture "Machine Learning", University of Freiburg, 2011.

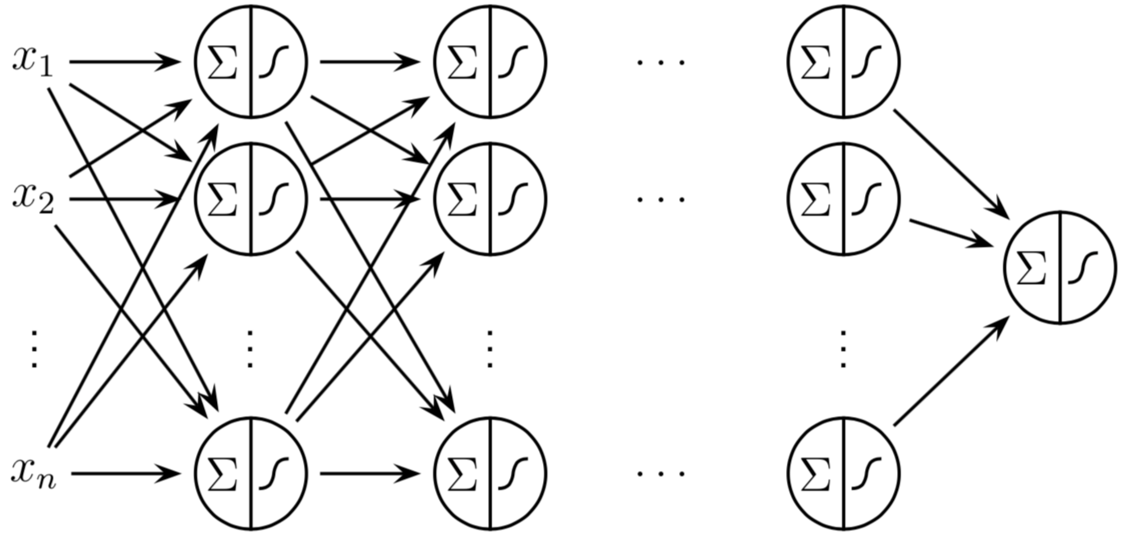
Training Multilayer Perceptrons

Training a Multilayer Perceptron

- Prerequisites:
  - training set $\mathcal{X} = \{(\mathbf{x}_1, t_1), \dots, (\mathbf{x}_N, t_N)\}$ with feature vectors $\mathbf{x}_i$ and target values $t_i, i = 1, \dots, N$.
  - t_i is a discrete class label (classification) or a real number (regression) but can also be a vector (*e.g.* one-hot encoding)
  - A multilayer perceptron with a suitable topology.
- Goal: Automatically adapt the weights of the MLP to map from an input feature vector to the desired output value.
- Idea: minimize an error term (loss function):

$$E(\mathbf{w}, \mathcal{X}) = \frac{1}{2} \sum_{i=1}^{N} ||y(\mathbf{x}_i, \mathbf{w}) - t_i||^2$$

- Here the error function is the sum of squared differences of the network output $y(\mathbf{x}_i, \mathbf{w})$ for input pattern $\mathbf{x}$, weight vector $\mathbf{w}$ and the target t_i .



Adapted from M. Riedmiller, Lecture “Machine Learning”, University of Freiburg, 2011.

Training a Multilayer Perceptron

- Idea: minimize an error term (loss function):

$$E(\mathbf{w}, \mathcal{X}) = \frac{1}{2} \sum_{i=1}^{N} ||y(\mathbf{x}_i, \mathbf{w}) - t_i||^2$$

- Here the error function is the sum of squared differences of the network output $y(\mathbf{x}_i, \mathbf{w})$ for input pattern $\mathbf{x}$, weight vector $\mathbf{w}$ and the target t_i .

- Learning in this context refers to computing the optimal weights $\hat{\mathbf{w}}$ that minimize the error function:

$$\hat{\mathbf{w}} = \arg\min_{\mathbf{w}} E(\mathbf{w}, \mathcal{X})$$

- $E(\mathbf{w}, \mathcal{X})$ is a multidimensional function depending on $\mathbf{w}$ and can be numerically analyzed to find the minimum.

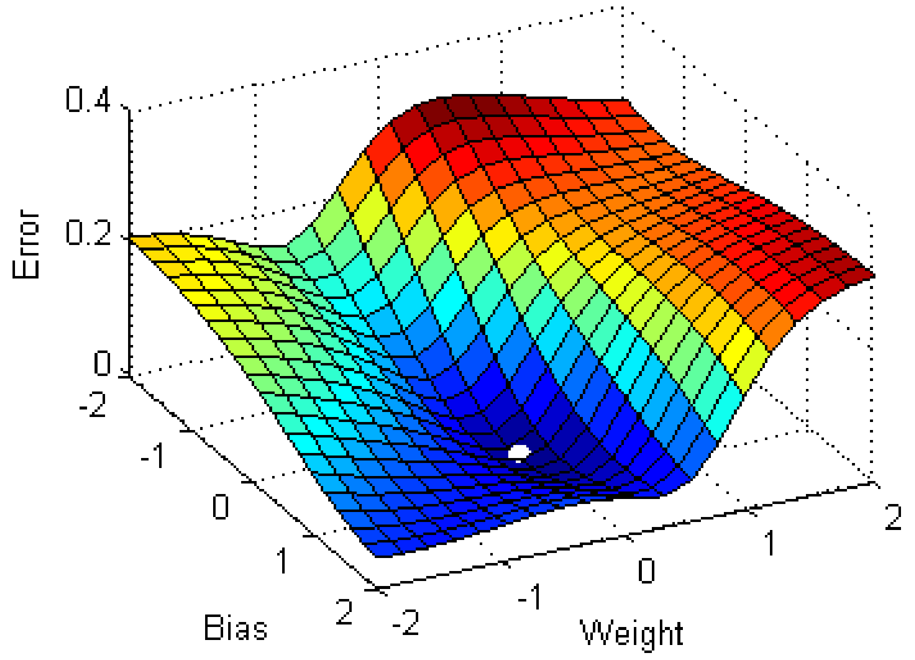


Image: https://nikcheerla.github.io/deeplearningschool//media/loss3d.gif

Optimization Theory

- Considering a simple 1D example, assume we're starting at a point $\mathbf{w}$. How do we identify a point $\mathbf{v}$ that satisfies: $f(\mathbf{v}) < f(\mathbf{w})$?

- Use the direction of the gradient to identify the next position.

- This works analogous in higher dimensions!

- General principle of updating the position towards the (local) minimum:

$$\underbrace{v_i}_{\text{New position}} = \overbrace{w_i}^{\text{Old position}} - \underbrace{\epsilon}_{\substack{\text{Learning rate} \\ \epsilon > 0}} \overbrace{\frac{\partial f}{\partial w_i}}^{\text{Partial derivative with respect to } w_i}$$

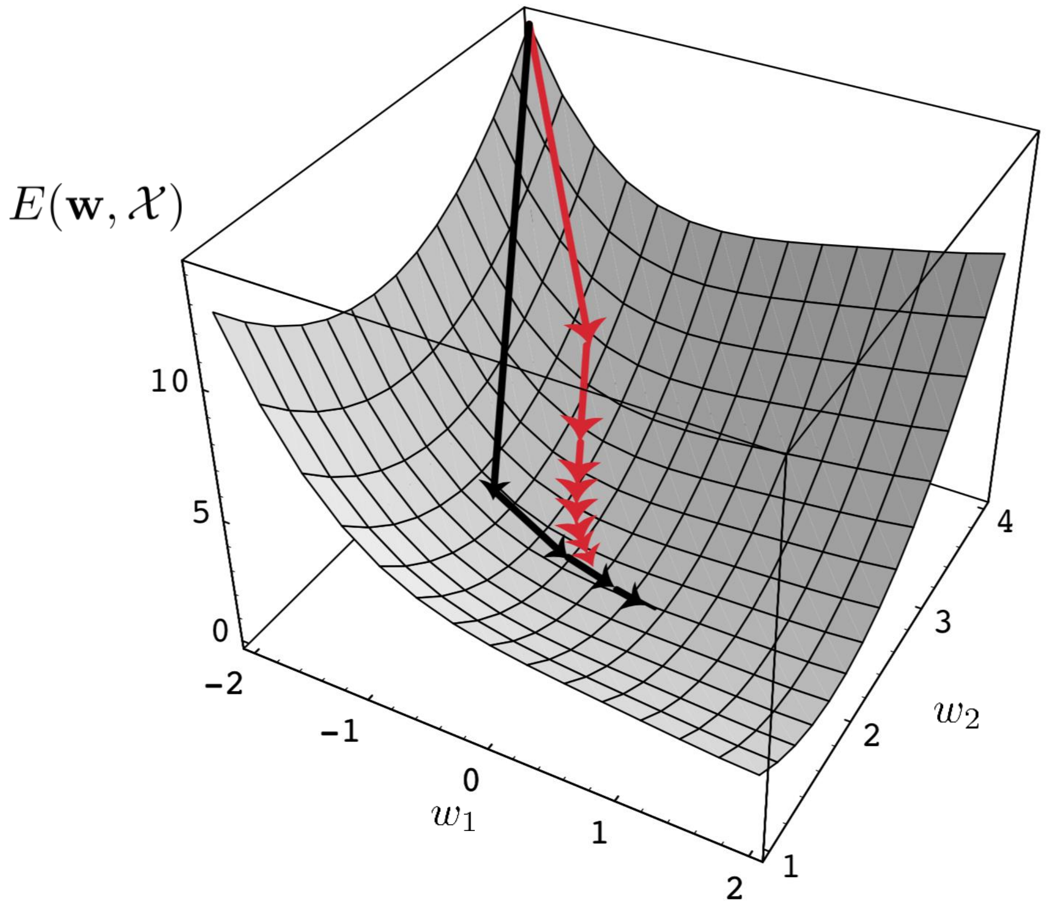


Image: R. Duda, P. Hart, D. Stork: Pattern Classification, John Wiley & Sons, 2001.

The Gradient Descent Algorithm

Input: a continuous and differentiable function $f(\cdot)$ and a learning rate ϵ .

Output: result vector that is (close to) a local minimum of $f(\cdot)$.

```
1. choose starting point
2. **while** $||\nabla f(\mathbf{w})||$ not close to zero **do**
3.     $\mathbf{w} = \mathbf{w} - \epsilon \nabla f(\mathbf{w})$
4. **end while**
5. **return** $\mathbf{w}$
```

Open questions:

- How do we choose the initial location?
- How do we set the learning rate ϵ?
- Convergence properties of the algorithm?

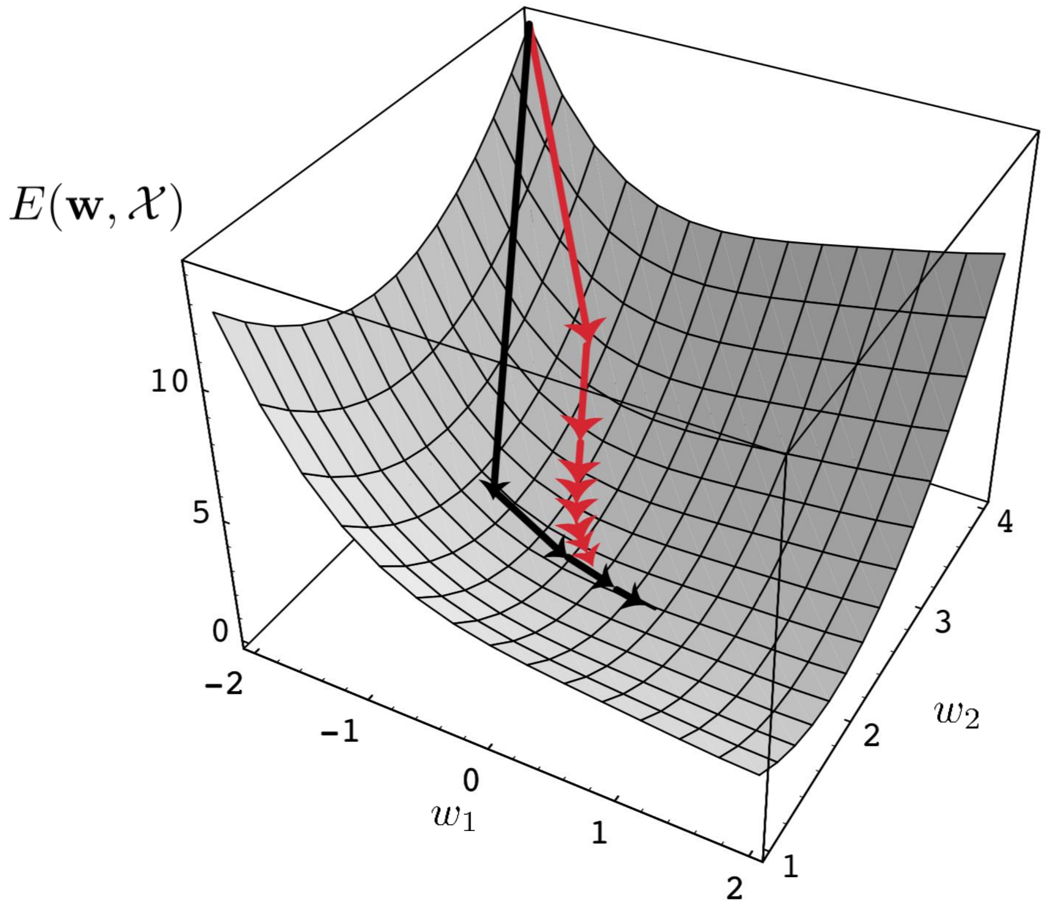


Image: R. Duda, P. Hart, D. Stork: Pattern Classification, John Wiley & Sons, 2001.

Selecting an Appropriate Learning Rate

- The algorithm converges if the learning rate is set sufficiently small.
- If ϵ is set too small, the algorithm converges but it might take very long.
- If ϵ is set too large, the algorithm may actually diverge.
- Other problems that may occur:
  - Large ϵ may be required to pass a flat area but the learning rate needs to be small to sneak into the minimum.
  - **Zig-zagging**: if multiple dimensions are considered, a single value for ϵ might not be appropriate for all dimensions.

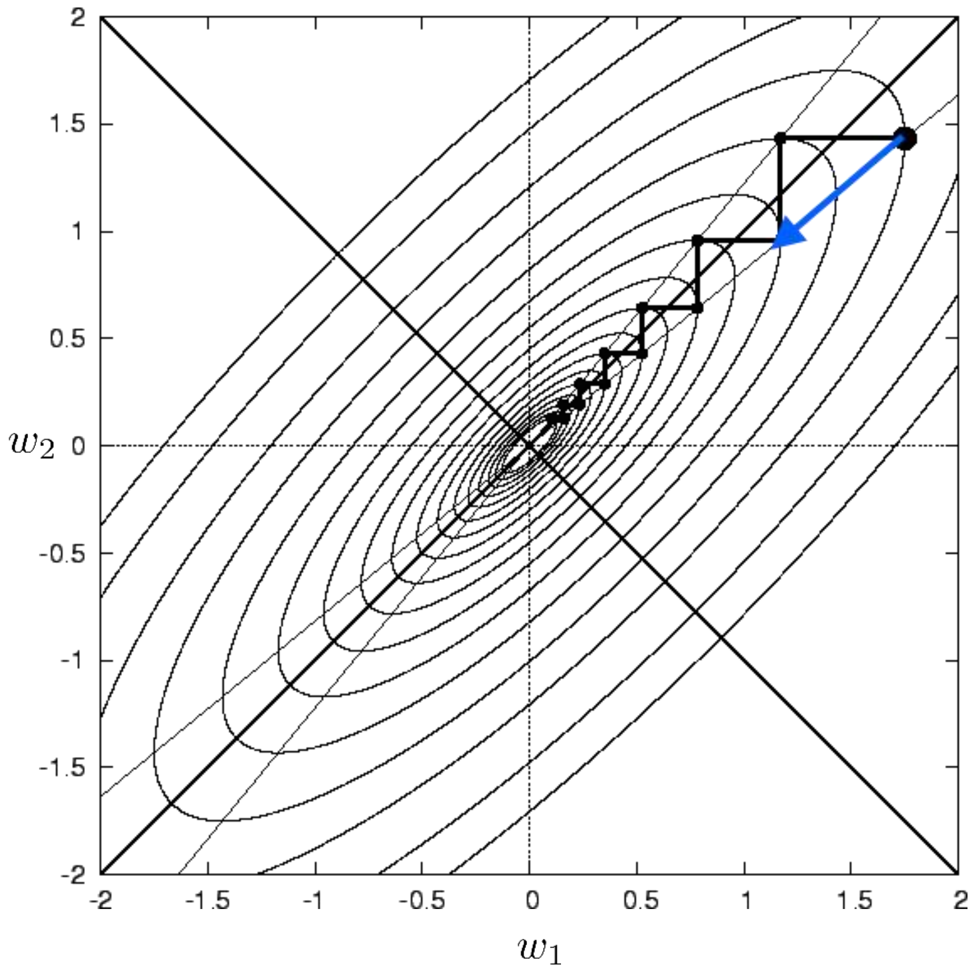


Image: https://i.stack.imgur.com/Kw20F.png

Backpropagation Algorithm Pseudocode

With $\frac{\partial E}{\partial w_{ij}}$, we now have all required components to finally train the network given our training data:

```
1.  Choose initial weight vector
2.  Initialize minimization approach
3.  **while** error did not converge **do**
4.      **for all** $(\mathbf{x}, \mathbf{t}) \in \mathcal{X}$ **do**
5.          forward pass with $\mathbf{x}$ to compute the
            network output
6.          calculate $\frac{\partial e(\mathbf{x})}{\partial w_{ij}}$ for all weights
7.      **end for**
8.      calculate $\frac{\partial E(\mathbf{x})}{\partial w_{ij}}$ for all weights and sum over
        all training samples (error backpropagation)
9.      perform weight update step of the
        optimization approach
10. **end while**
```

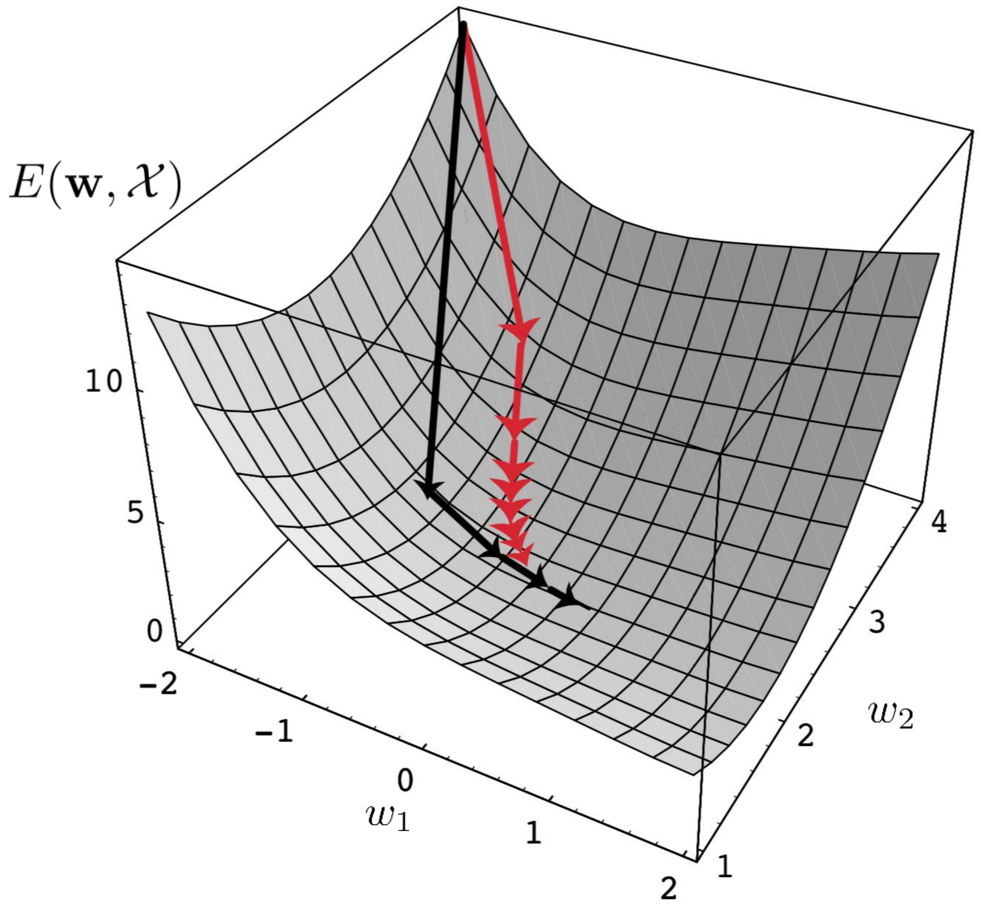
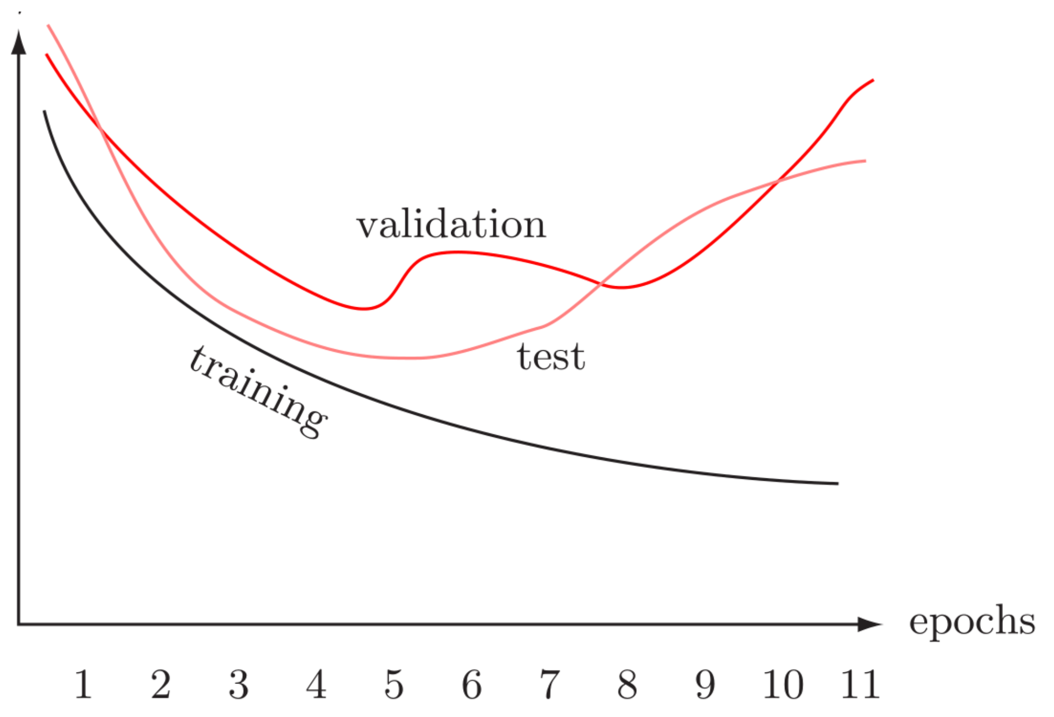


Image: R. Duda, P. Hart, D. Stork: Pattern Classification, John Wiley & Sons, 2001.

Backpropagation Algorithm Pseudocode

- Learning strategies:
  - **Learning by pattern**: only one training pattern is considered for one update step of function minimization (only for vanilla gradient descent).
  - **Learning by mini-batch**: subset of the training data (mini-batch) is used for performing the update.
  - **Learning by epoch**: all training patterns are considered for one update step of the function minimization.

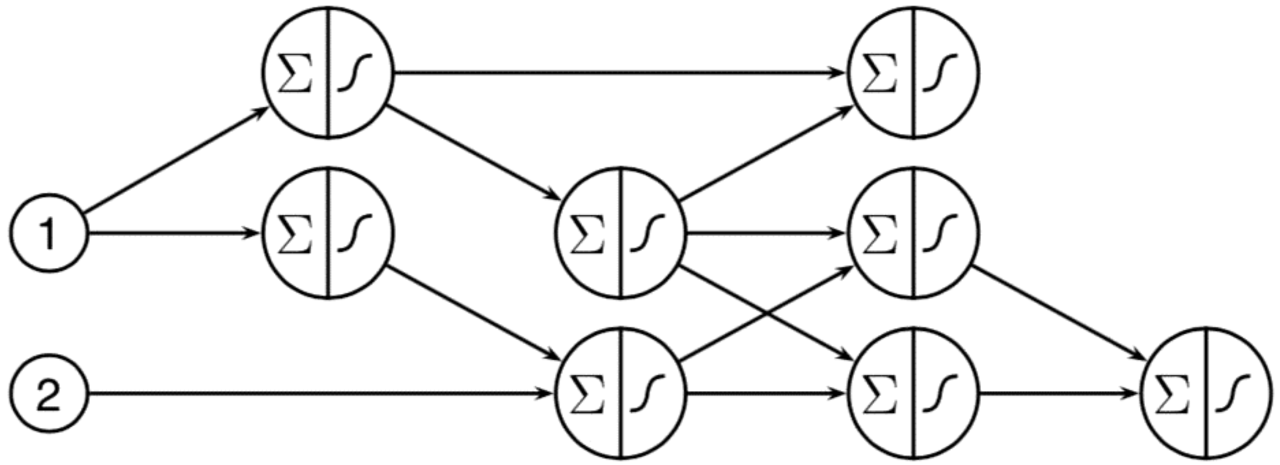
- Use three splits of the data, namely:
  - **Training set**: this data set is used to train the model.
  - **Validation set**: this subset of the data is used to identify potential overfitting.
  - **Test set**: the final performance is evaluated on the test set. No data of the test set should be part of the training and validation!



Although the training error keeps steadily decreasing, validation and test error increase, *i.e.*, the model is overfitting the data.

Image: R. Duda, P. Hart, D. Stork: Pattern Classification, John Wiley & Sons, 2001.

Backpropagation Algorithm Animation



Adapted from M. Riedmiller, Lecture “Machine Learning”, University of Freiburg, 2011.

Example: MNIST

- Automatic classification of handwritten digits (*e.g.*, for automatic address recognition on letters).

- Training data consists of 28x28 pixel sized image snippets with known classification (classes 0-9).

- The multilayer perceptron used in this case ...
  - … has 784 (28x28) input nodes
  - … two hidden layers with 300 and 100 hidden nodes, respectively
  - … 10 output nodes, one node for each of the classes
  - … was trained using backpropagation.

- If you want to try it on your own, the website is: http://scs.ryerson.ca/~aharley/vis/fc/

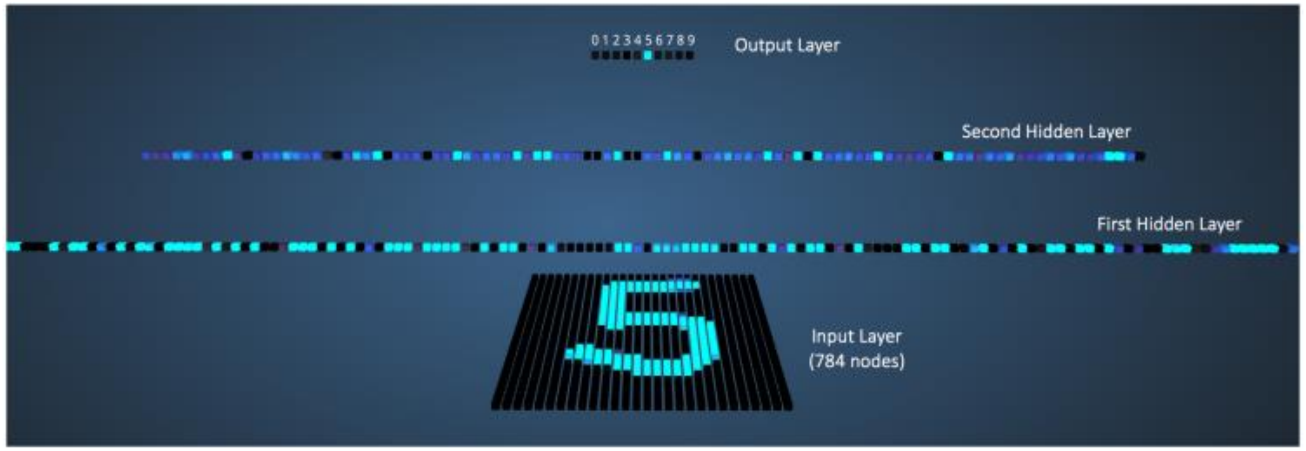
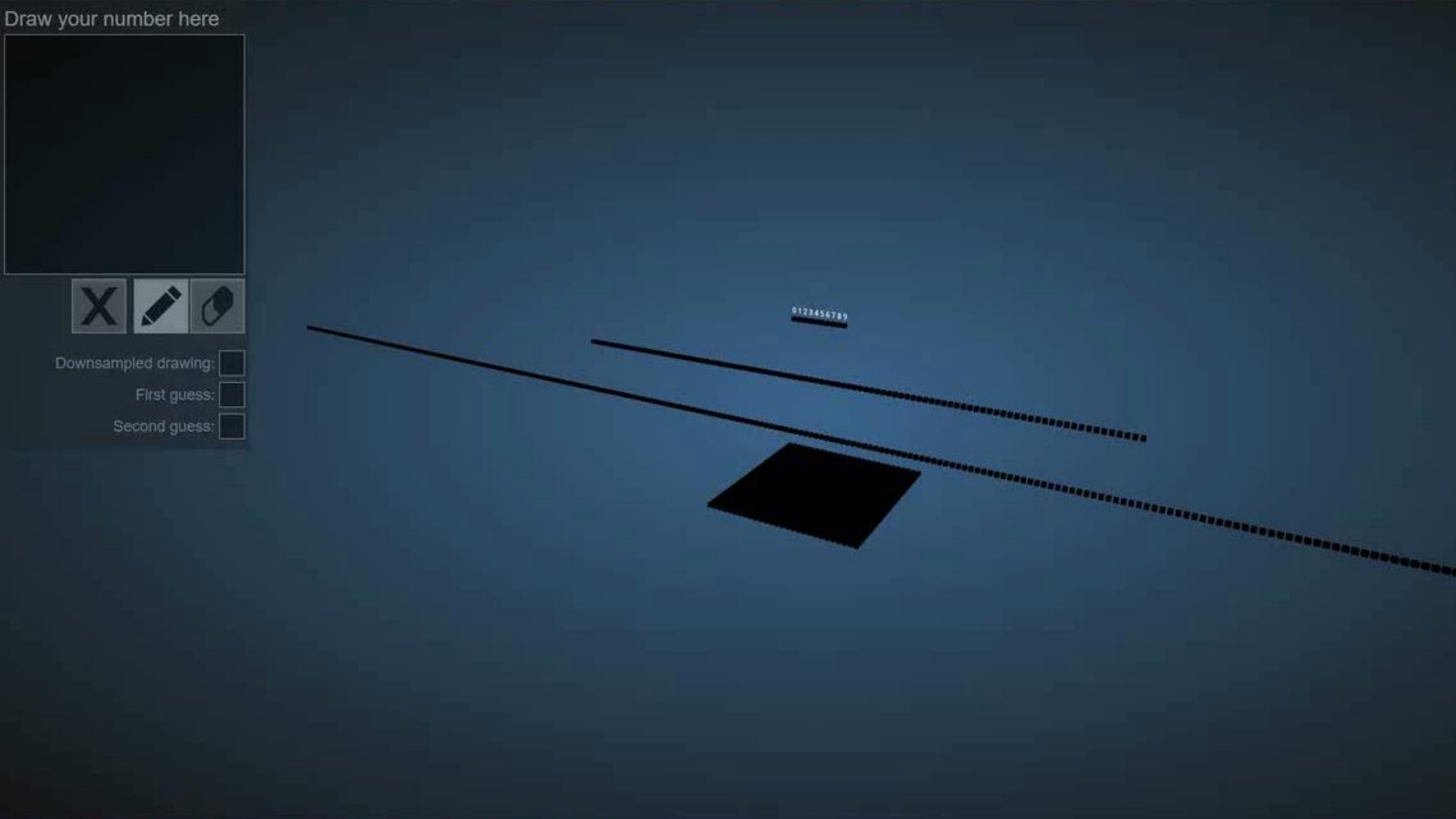


Image: https://ujjwalkarn.me/2016/08/09/quick-intro-neural-networks/


Draw your number here
Downsampled drawing:
First guess:
Second guess:
0123456789

Convolutional Neural Networks

What are Convolutional Neural Networks?

- “Classical” machine learning approach:
  - Feature Extraction
  - Feature Selection
  - Classification
- Approach pursued by Convolutional Neural Networks:
  Learn the features as well using the data!!

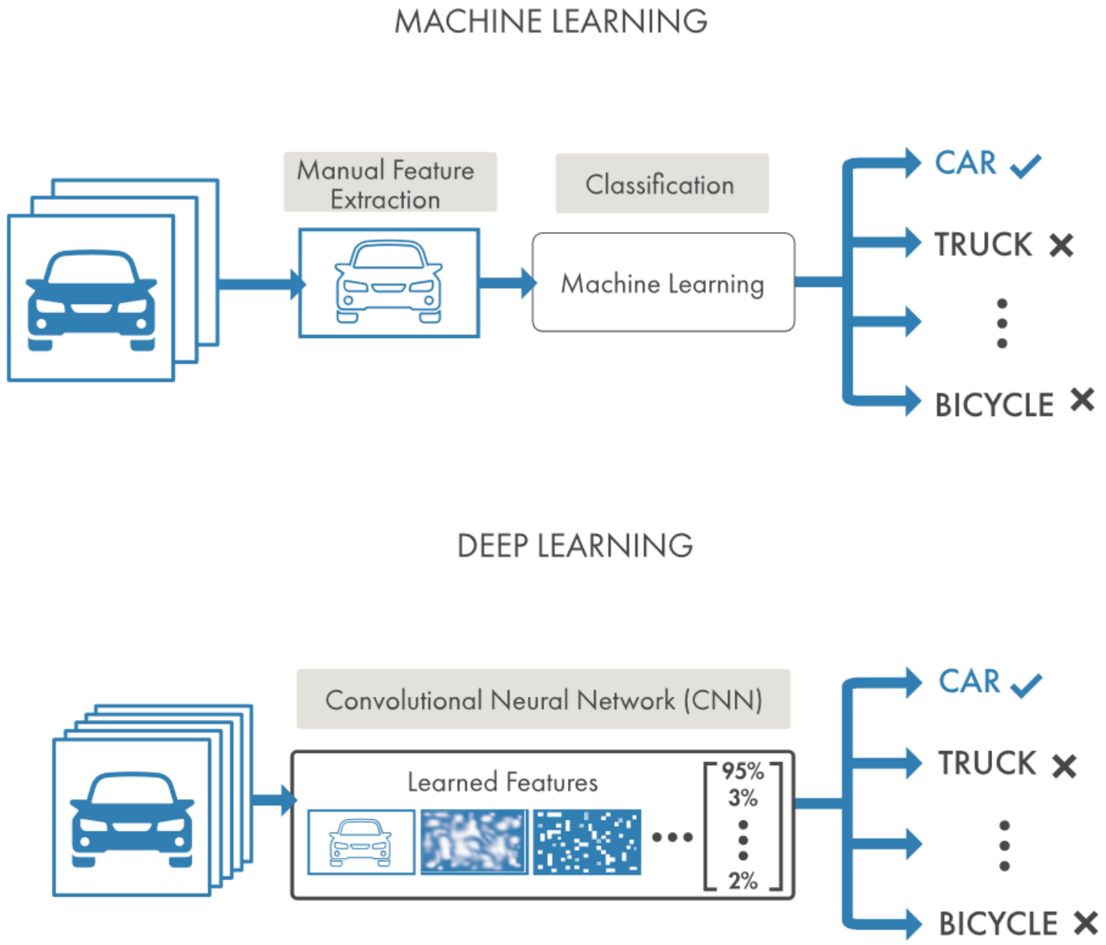


Image: https://de.mathworks.com/discovery/deep-learning.html

What are Convolutional Neural Networks?

- “Classical” machine learning approach:
  - Feature Extraction
  - Feature Selection
  - Classification
- Approach pursued by Convolutional Neural Networks:
  Learn the features as well using the data!!
- Major ideas and improvements in the following fields enabled the successful use of deep neural networks:
  - New activation functions
  - Regularization methods
  - Data augmentation
  - Optimization techniques
- Availability of large training data sets (*e.g.*, ImageNet)
- Powerful GPGPUs significantly accelerated the training process (multiple times faster than training on CPUs)

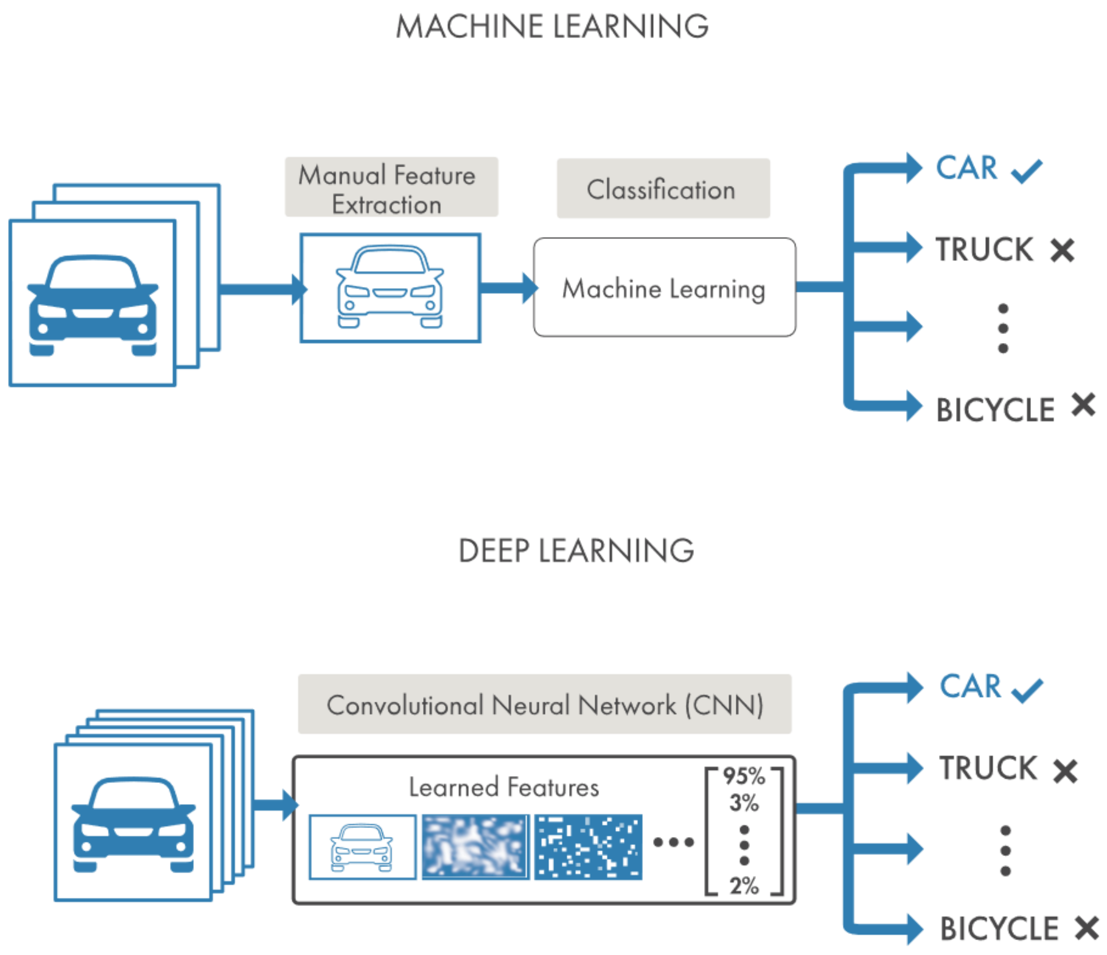
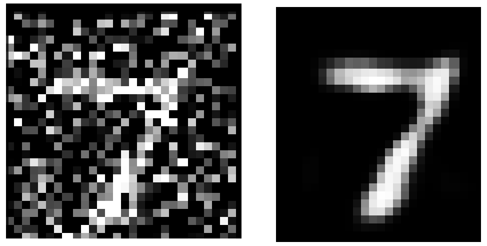
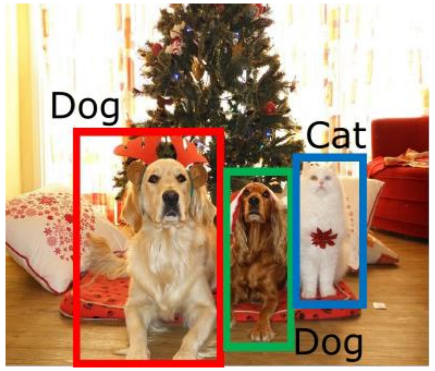


Image: https://de.mathworks.com/discovery/deep-learning.html

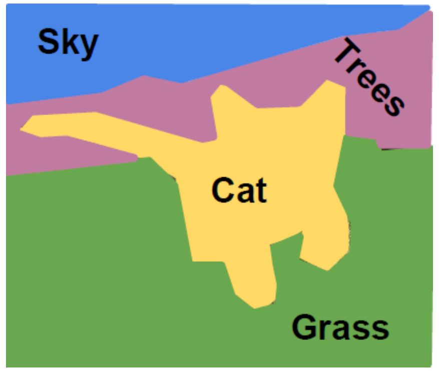
What Kind of Tasks Can be Solved by CNNs?



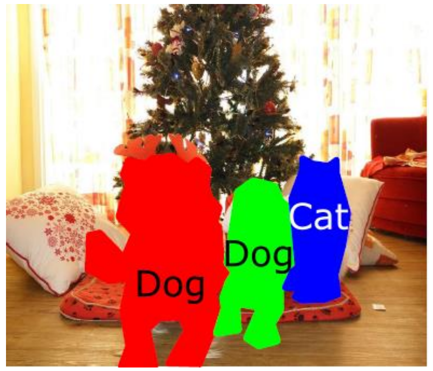
Denoising and signal enhancement



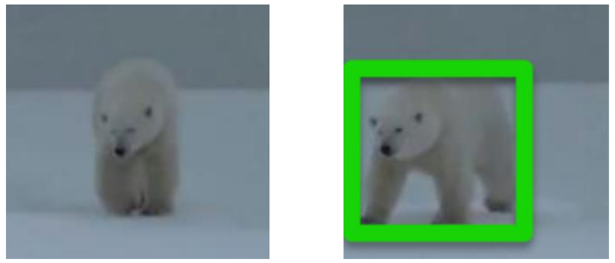
Object detection and classification



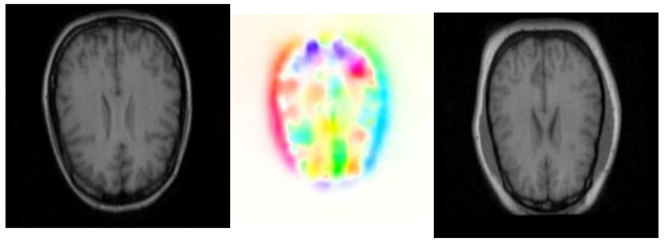
Semantic segmentation



Instance segmentation



Object tracking



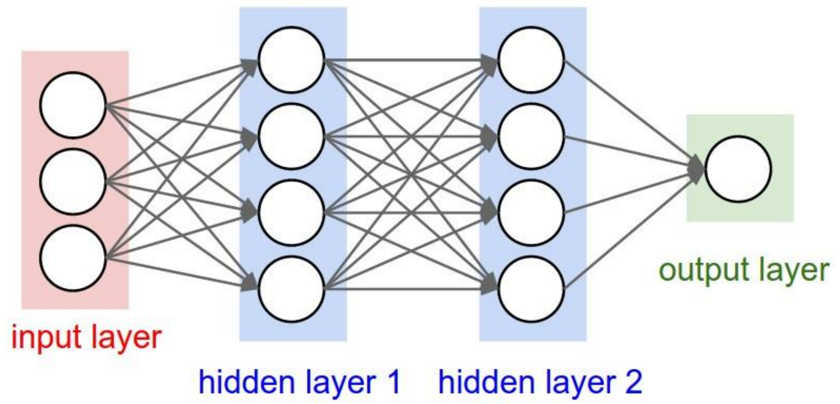
Registration

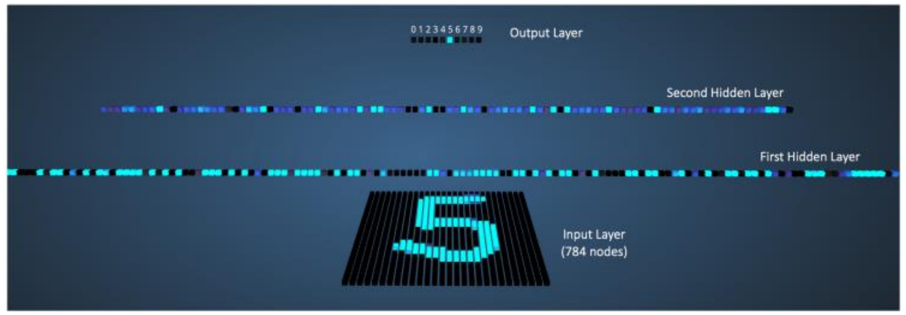
Image: Held et al., 2016, CS231n, 2018

Can’t we use multilayer perceptrons?

- Consider the MNIST MLP example again, we have:
  - Input images of size 28x28 pixels
  - 300 neurons in the 1st hidden layer
  - 100 neurons in the 2nd hidden layer
  - 10 neurons in the output layer
- As we have a fully connected architecture, we have:
  - $28 \cdot 28 \cdot 300 + 300 = 235500$ weights in the first layer
  - $300 \cdot 100 + 100 = 30100$ weights in the second layer
  - $10 \cdot 100 + 10 = 1010$ weights in the output layer
  - Which amounts to 266610 weights we have to train!
- While this amount still could be doable, it becomes infeasible for larger images.
- The same network applied to a 256x256 pixel sized image would already require almost 2 million weights to be adapted.

→ Fully-connected networks don’t scale well to full images and quickly lead to overfitting!

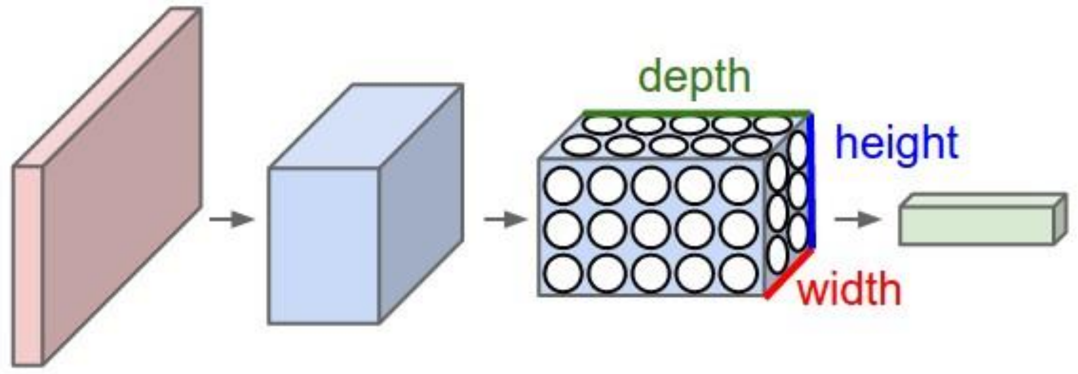




Images: http://cs231n.github.io/convolutional-networks/, http://scs.ryerson.ca/~aharley/vis/fc/

What are Deep Neural Networks?

- The main idea of *deep* neural networks is to arrange neurons of a layer in three dimensions (width, height, depth).
- Note: Depth in this context refers to the number of filters per layer, not (only) the total number of layers.
- The architecture explicitly makes use of the nature of images:
  - Local receptive fields (each neuron only “looks” at a local region of the image).
  - Shared weights (properties like edges or corners are interesting irrespective of the spatial location).
  - Down-sampling / pooling (objects in images can appear in different scales and at different locations).
  - Neighboring pixels are likely to contain similar/related information and should thus be processed similarly.



Layer arrangement in a convolutional neural network (CNN) as 3D volumes.

Building Blocks of Convolutional Neural Networks

- The most-used building blocks of convolutional neural networks are:
  - **Input Layer** (the raw pixel values, potentially including color channels)
  - **Convolutional Layer** (learned filters that transform the input images to filtered images. Weights are shared among neurons of one filter and each neuron computes a dot product of it's weights with a small image region it is connected to).
  - **Activation Layer** (similar to activation in MLPs by an elementwise application of the activation function)
  - **Pooling Layer** (Down-sampling of the feature maps)
  - **Fully-connected Layer** (identical to what we learned for MLPs, *i.e.*, each neuron is connected to all predecessors. Mostly used as the output layer with a neuron for each category)

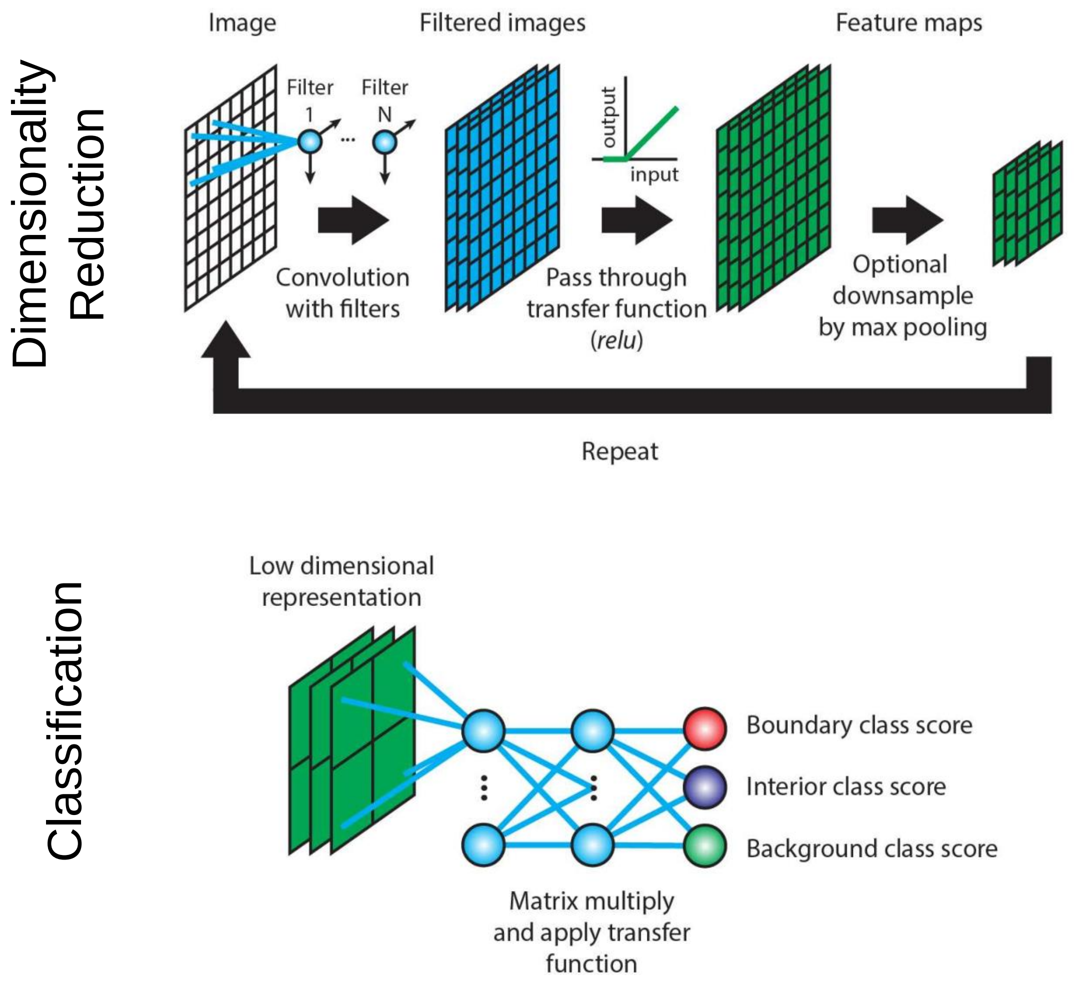


Image: van Valen *et al.*, 2016.

Building Blocks of Convolutional Neural Networks

- As for MLPs, convolutional neural networks transform an input image layer by layer from pixel values to class scores.
- The results of the convolutional and the fully-connected layers depend on the **weights and biases** of the neurons and **must be trained**.
- Activation layers and pooling layers implement a **fixed function**, *i.e.*, they require **no parameter training**.
- Convolutional, pooling and fully connected layers have additional **hyperparameters**:
  - Receptive field / filter size
  - Number of hidden neurons
  - Stride
  - Border handling
- We'll get to all of this in detail later on!

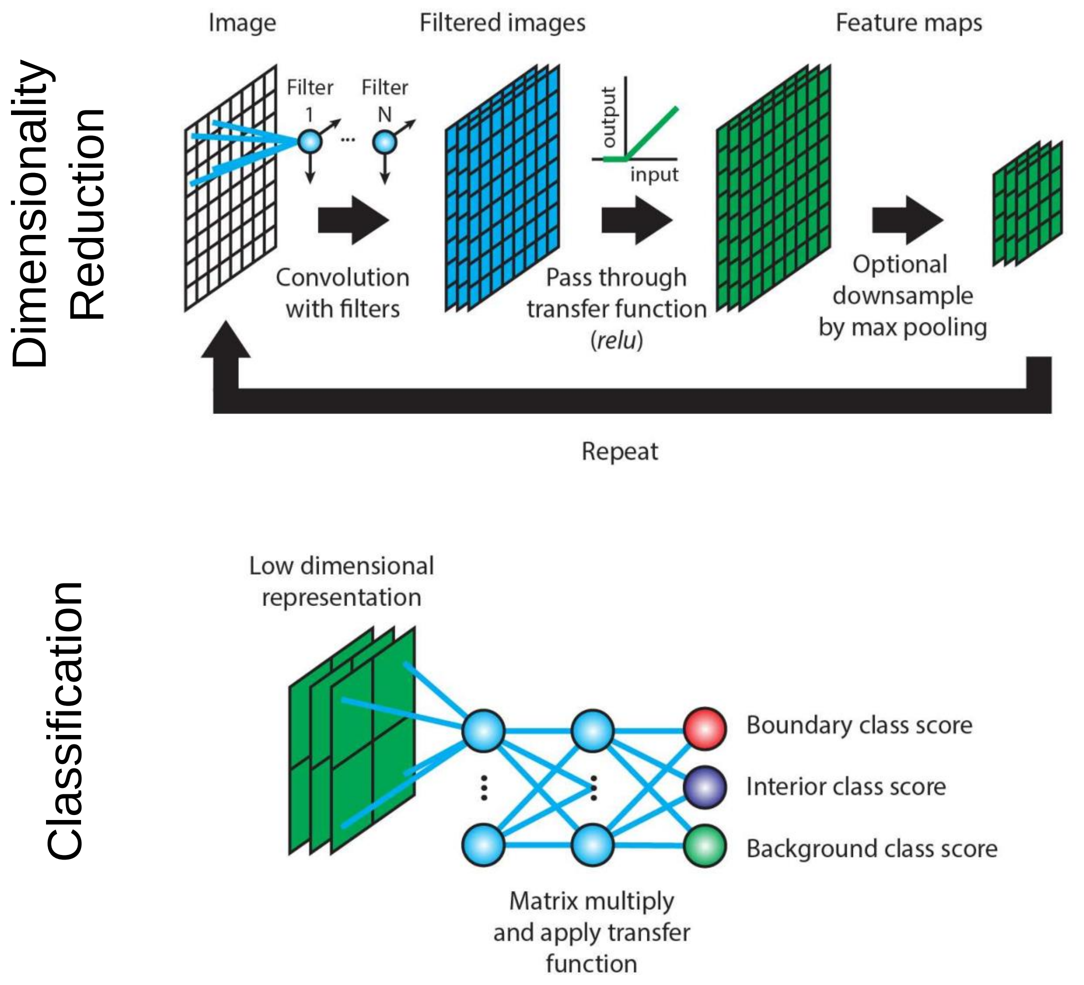
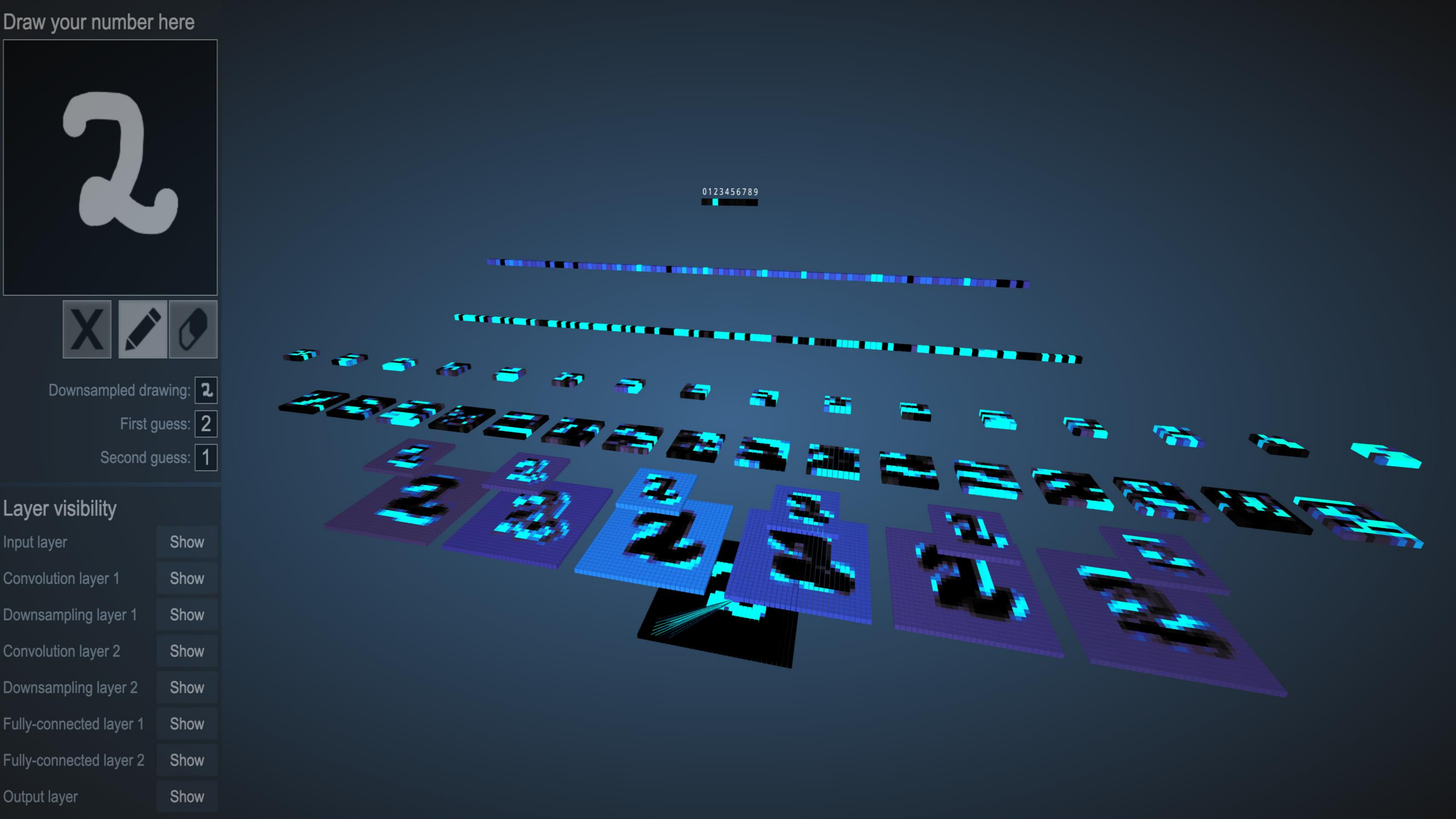
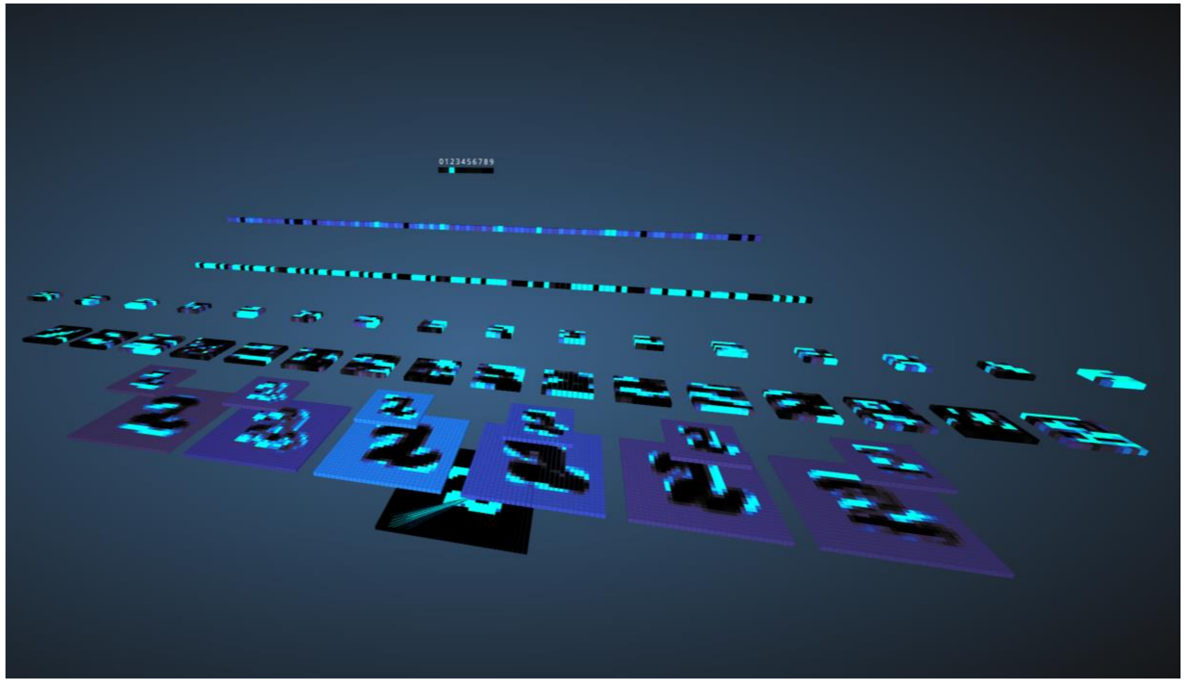


Image: van Valen *et al.*, 2016.


Draw your number here
Downsampled drawing:
First guess: 2
Second guess: 1
Layer visibility
Input layer Show
Convolution layer 1 Show
Downsampling layer 1 Show
Convolution layer 2 Show
Downsampling layer 2 Show
Fully-connected layer 1 Show
Fully-connected layer 2 Show
Output layer Show
0123456789

Example: Convolutional Neural Network for the MNIST Data Set

- An exemplary CNN to classify the MNIST data set might be comprised of the following layers (LeNet-5 by Yann LeCun, 1998):
  - Input Layer (32x32x1)
  - Convolutional Layer 1 (28x28x6)
  - Max Pooling Layer 1 (14x14x6)
  - Convolutional Layer 2 (10x10x16)
  - Max Pooling Layer 2 (5x5x16)
  - Fully-connected Layer 1 (1x1x120)
  - Fully-connected Layer 2 (1x1x100)
  - Output Layer (1x1x10)
- Already 99% accuracy achieved by this network!



Visualization of the LeNet-5 by Yann LeCun et al., 1998.

Image: http://scs.ryerson.ca/~aharley/vis/conv/

Institute of Imaging and Computer Vision RWTH AACHEN UNIVERSITY

Core Building Block: Convolutional Layers

- A convolutional layer consists of **multiple learnable filters**.
- Each filter has a **small spatial receptive field** but spans the full depth of the input volume (*e.g.*, 5x5x3, height and width and a depth of 3 for color channels).
- In the forward pass through the network, the input volume is convolved with each of the filters (dot product of the filter with all spatial input locations)
- The intuition is that the **network will learn filters** that activate upon "seeing" visual features like edges, orientations, colors, …
- Each filter produces an **activation map,** and those maps are stacked together to form the output volume.

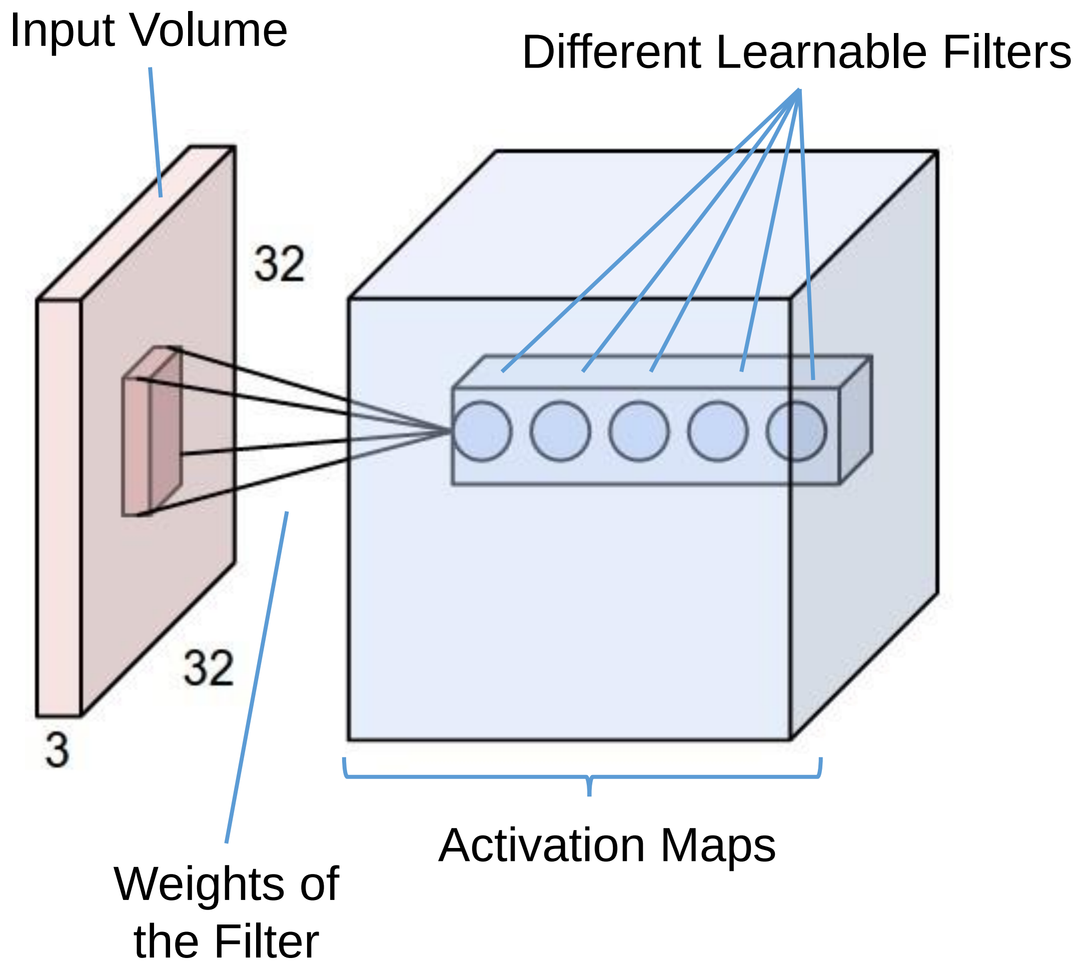


Image: http://cs231n.github.io/convolutional-networks/

Core Building Block: Convolutional Layers

- If interpreted as a neural network again, each entry of a 3D output volume is the output of a neuron with a small receptive field.
- All neurons of one activation map share the same weights (*i.e.*, the same filter is applied on the entire image).
- In contrast to fully-connected architectures, we thus have a **dramatic reduction of parameters**.
- The hyperparameter controlling the spatial extent of the convolutional filters is called **receptive field**.
- Connections are local in space but extend along the entire depth of the input volume.

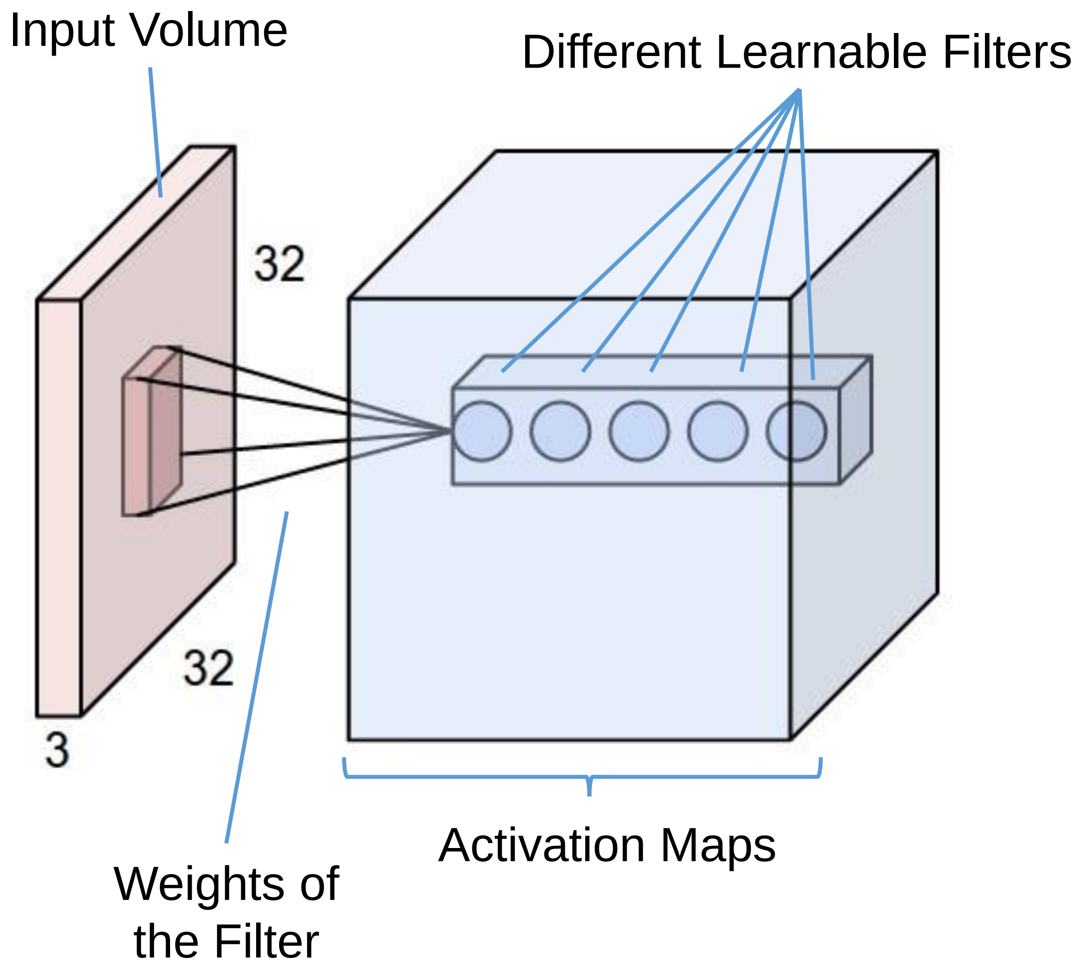


Image: http://cs231n.github.io/convolutional-networks/

Core Building Block: Convolutional Layers

- Example 1:
  - Input volume has size 32x32x3, (*e.g.* an RGB CIFAR-10 image).
  - Receptive field (or the filter size) is 5x5
  - Each neuron in the convolutional layer will have weights to a 5x5x3 region in the input volume
  - Total number of 5*5*3 = 75 weights (and +1 bias parameter) shared by all neurons.
  - Notice that the extent of the connectivity along the depth axis must be 3, since this is the depth of the input volume.
- Example 2:
  - Input volume of size 16x16x20.
  - Receptive field 3x3
  - Each neuron in the convolutional layer would now have a total of 3*3*20 = 180 (+1 for bias) connections to the input volume.
  - Notice that, again, the connectivity is local in space (e.g. 3x3), but full along the input depth (20).

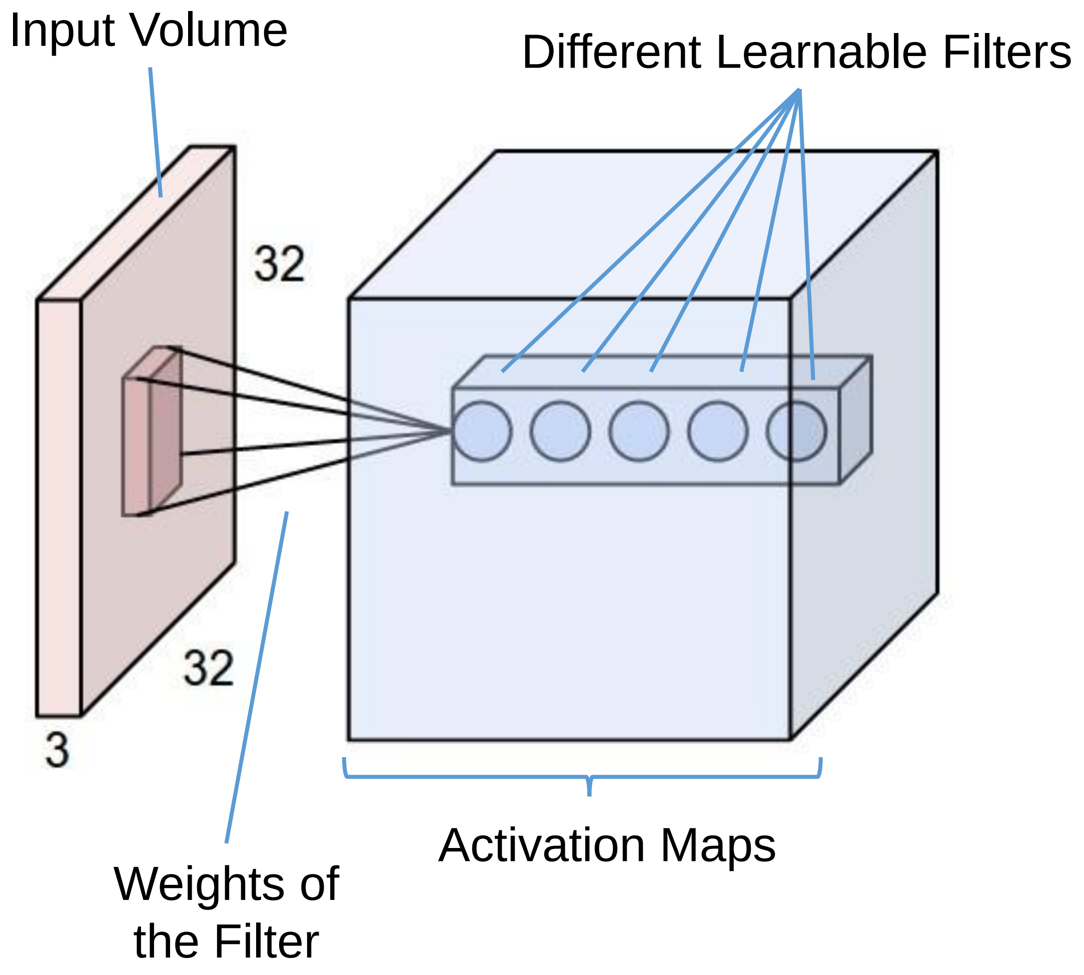
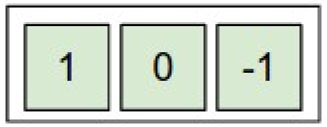


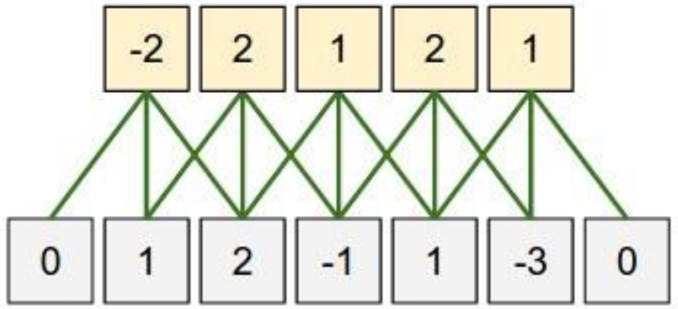
Image: http://cs231n.github.io/convolutional-networks/

Spatial Arrangement of the Neurons

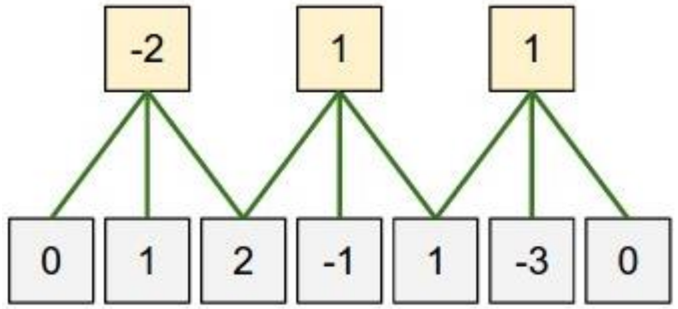
- How many neurons are contained in each output volume? How are they arranged?
- The size of the output volume is determined by three hyperparameters, namely **depth, stride** and **zero-padding**.
- **Depth** (K): the number of different filters we want to use.
- **Stride** (S): step size used for the convolution. With stride of 1 each pixel is visited, stride of 2 only every 2nd pixel, …
- **Zero-Padding** (P): to preserve the spatial size of the input volume, (zero-)padding can be used to pad the input volume at the border regions.
- Moreover, the **receptive field** (F) determines the size of the filter (*e.g.*, 3x3).

Filter Size: 3x1





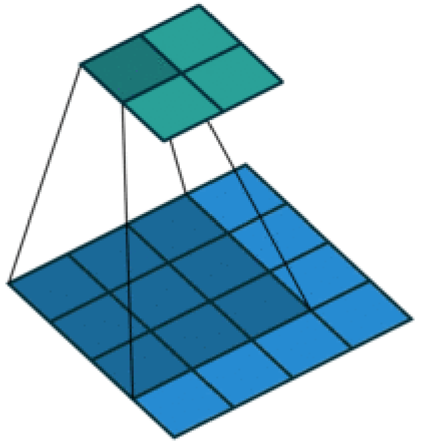
Stride = 1, Zero-Padding



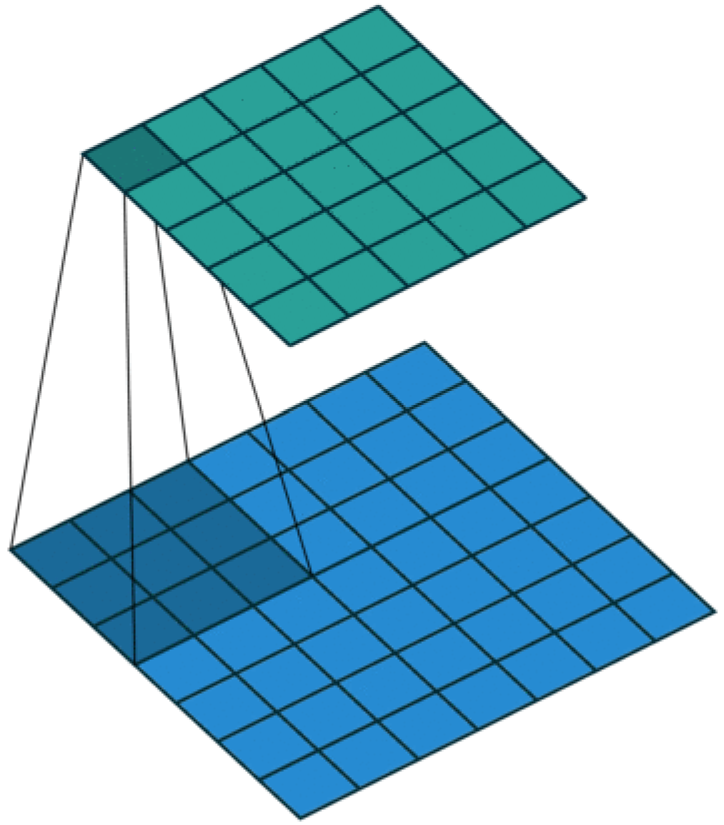
Stride = 2, Zero-Padding

Image: http://cs231n.github.io/convolutional-networks/

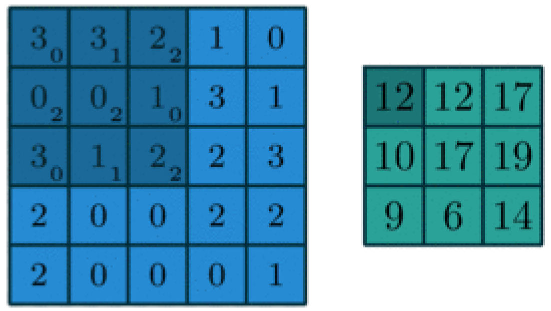
Spatial Arrangement of the Neurons



Receptive Field: 3x3
Stride: 1
Padding: None



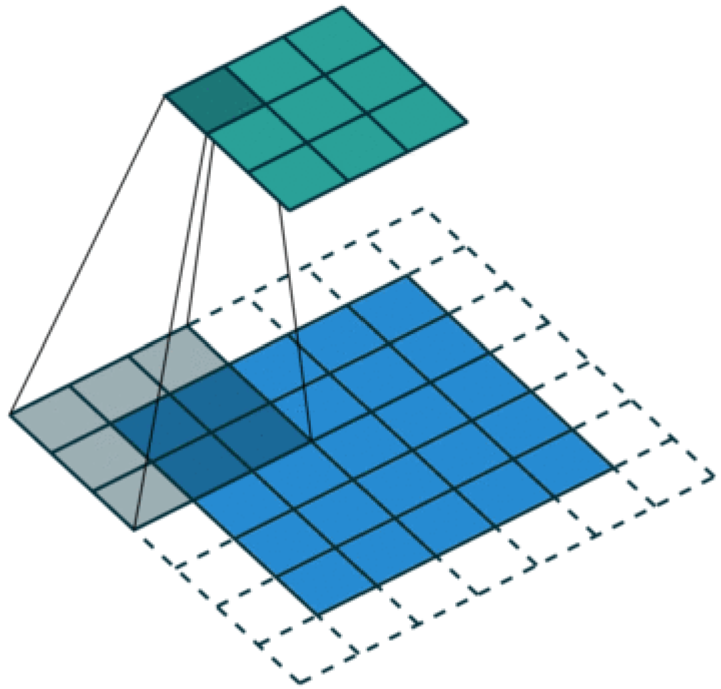
Receptive Field: 3x3
Stride: 1
Padding: None



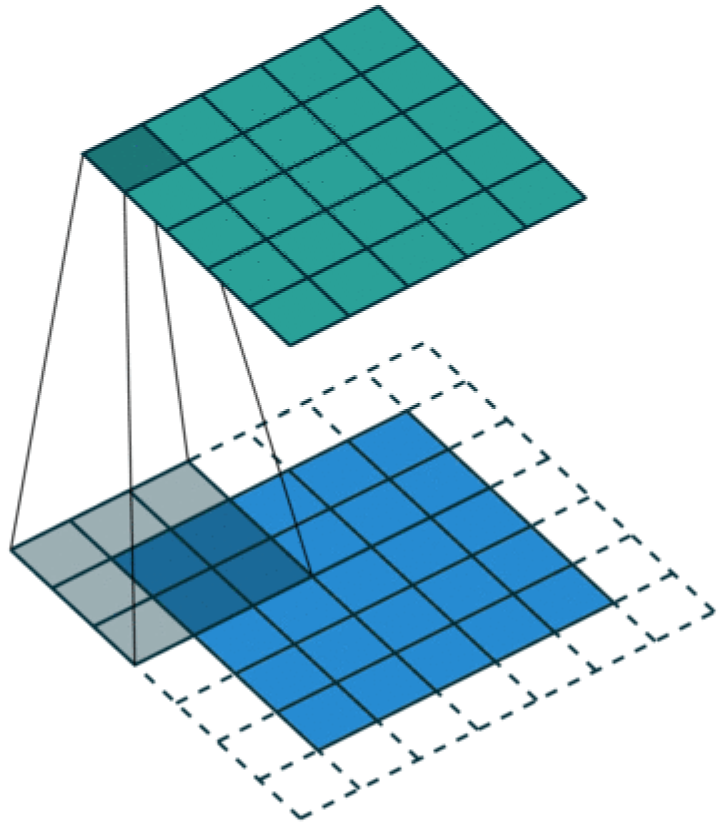
Receptive Field: 3x3
Stride: 1
Padding: None

Images: http://deeplearning.net/software/theano_versions/0.9.X/_images/

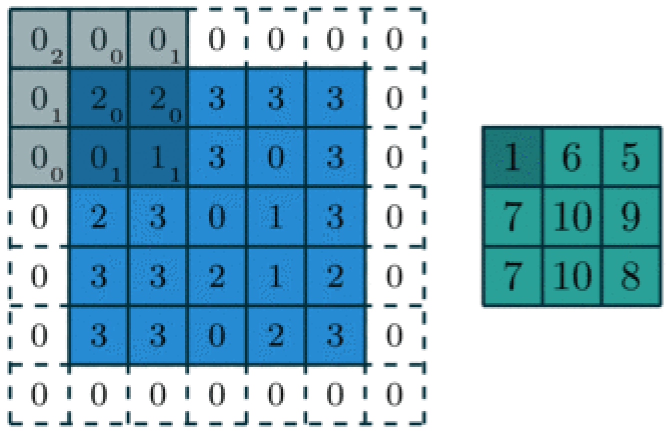
Spatial Arrangement of the Neurons



Receptive Field: 3x3
Stride: 2
Padding: 1



Receptive Field: 3x3
Stride: 1
Padding: 1



Receptive Field: 3x3
Stride: 2
Padding: 1

Images: http://deeplearning.net/software/theano_versions/0.9.X/_images/

Constraints for the Hyperparameters

- Hyperparameters need to be selected carefully, such that a discrete number of neurons can be used!
- Example:
  - Input width $W = 10$
  - No zero-padding, i.e., $P = 0$
  - Filter size $F = 3$
  - Problem: Impossible to use stride $S = 2$, as the neurons cannot be arranged symmetrically across the input.
  - Possible solution: use valid stride or add zero-padding.

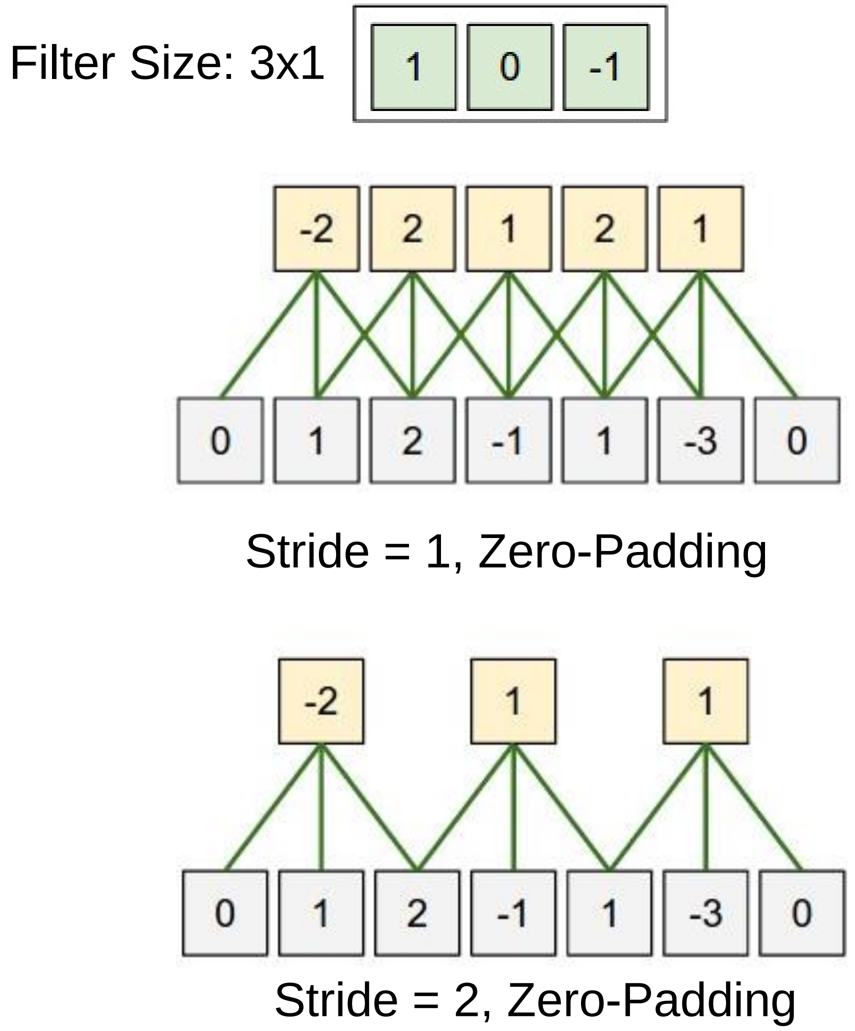


Image: http://cs231n.github.io/convolutional-networks/

Example: Convolution Demo

- Input image: 7x7x3 pixels
- Convolutional layer with two filters.
- Hyperparameters of the convolutional layer:
  - Depth: $K = 2$
  - Stride: $S = 2$
  - Padding: $P = 1$
  - Receptive field: $F = 3$
- Filter mask is iteratively scanned over the input volume according to the hyperparameters.
- At each location, the dot product between the filters and the input image is computed and the bias is added to obtain the output.

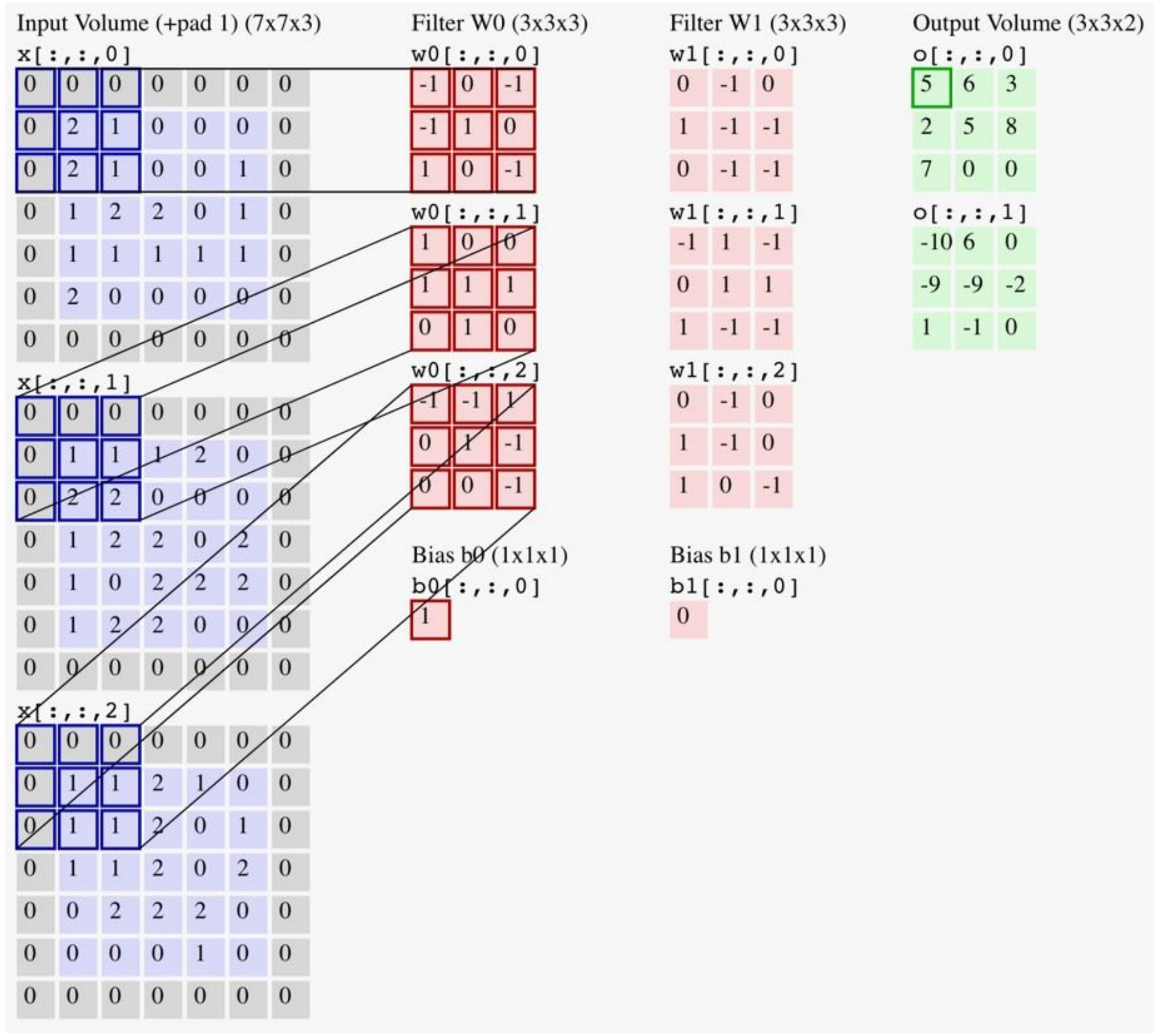
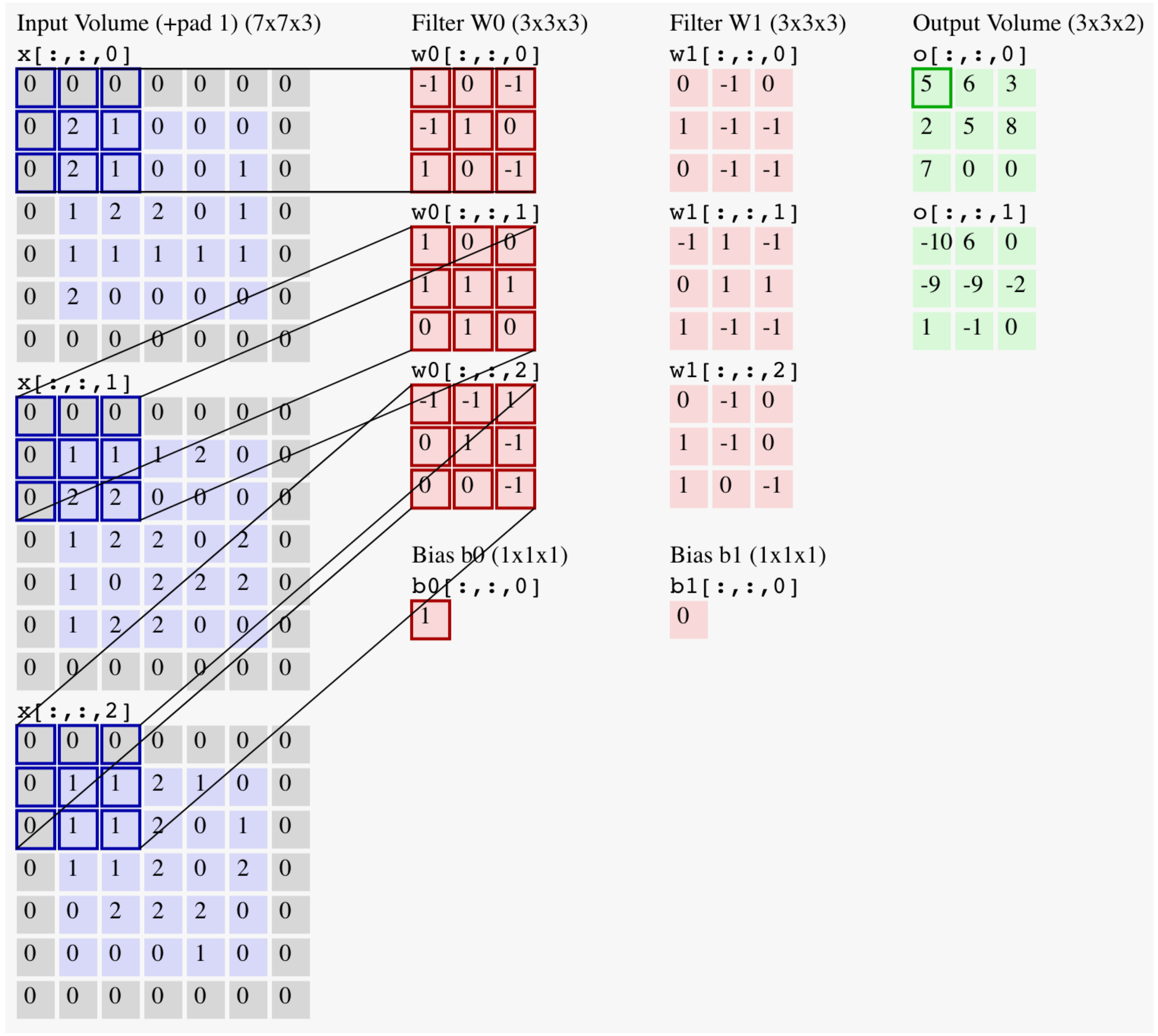


Image: http://cs231n.github.io/convolutional-networks/

Example: Convolution Demo

- Input image: 7x7x3 pixels
- Convolutional layer with two filters.
- Hyperparameters of the convolutional layer:
  - Depth: $K = 2$
  - Stride: $S = 2$
  - Padding: $P = 1$
  - Receptive field: $F = 3$
- Filter mask is iteratively scanned over the input volume according to the hyperparameters.
- At each location, the dot product between the filters and the input image is computed and the bias is added to obtain the output.



Input Volume (+pad 1) (7x7x3)

x[:,:,0]

0	0	0	0	0	0	0
0	2	1	0	0	0	0
0	2	1	0	0	1	0
0	1	2	2	0	1	0
0	1	1	1	1	1	0
0	2	0	0	0	0	0
0	0	0	0	0	0	0

x[:,:,1]

0	0	0	0	0	0	0
0	1	1	1	2	0	0
0	2	2	0	0	0	0
0	1	2	2	0	2	0
0	1	0	2	2	2	0
0	1	2	2	0	0	0
0	0	0	0	0	0	0

x[:,:,2]

0	0	0	0	0	0	0
0	1	1	2	1	0	0
0	1	1	2	0	1	0
0	1	1	2	0	2	0
0	0	2	2	2	0	0
0	0	0	0	1	0	0
0	0	0	0	0	0	0

Filter W0 (3x3x3)

w0[:,:,0]

-1	0	-1
-1	1	0
1	0	-1

w0[:,:,1]

1	0	0
1	1	1
0	1	0

w0[:,:,2]

-1	-1	1
0	1	-1
0	0	-1

Bias b0 (1x1x1)

b0[:,:,0]

1

Filter W1 (3x3x3)

w1[:,:,0]

0	-1	0
1	-1	-1
0	-1	-1

w1[:,:,1]

-1	1	-1
0	1	1
1	-1	-1

w1[:,:,2]

0	-1	0
1	-1	0
1	0	-1

Bias b1 (1x1x1)

b1[:,:,0]

0

Output Volume (3x3x2)

o[:,:,0]

5	6	3
2	5	8
7	0	0

o[:,:,1]

-10	6	0
-9	-9	-2
1	-1	0

Image: http://cs231n.github.io/convolutional-networks/

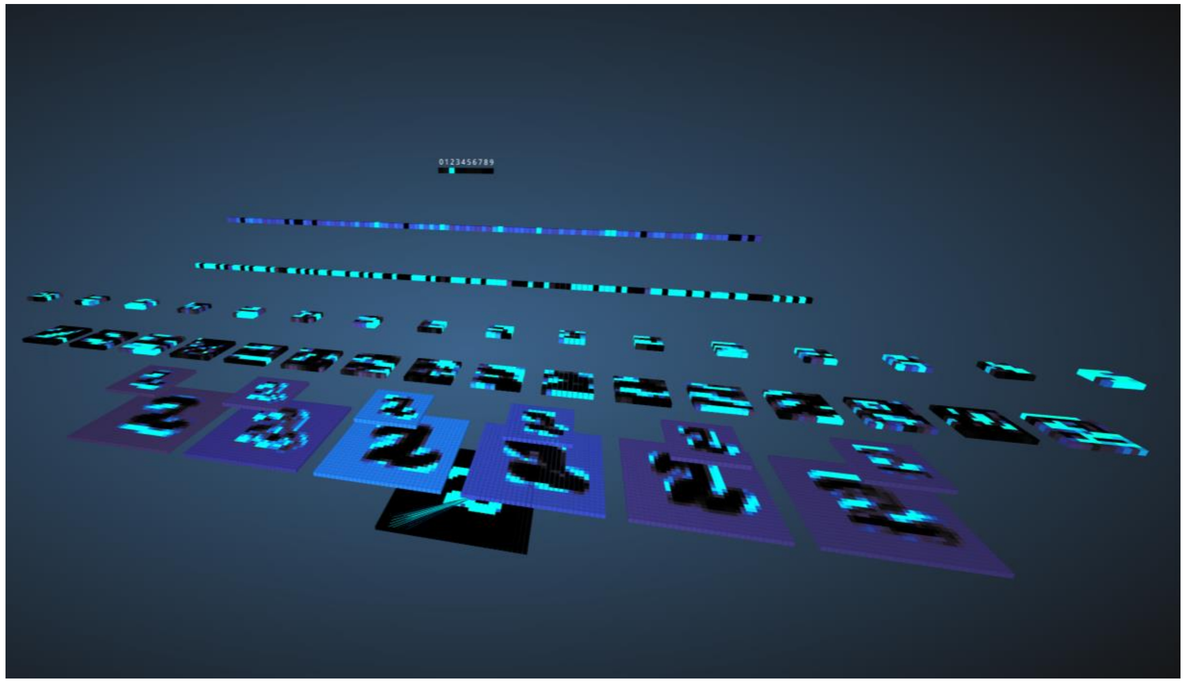
Why do we use parameter sharing?

- Assumption: if a feature is useful to compute at some spatial position, then it should also be useful to compute it at a different position.
- Thus, there is no need to relearn to detect the same feature at every image location.
- Considering again the MNIST example:
  - Hyperparameters: $K = 6, F = 5, S = 1, P = 0$
  - Using a separate weights for each neuron with local 5x5 connectivity, the first layer would have:

$$(5 \cdot 5 + 1) \cdot (28 \cdot 28) \cdot 6 = 122304$$

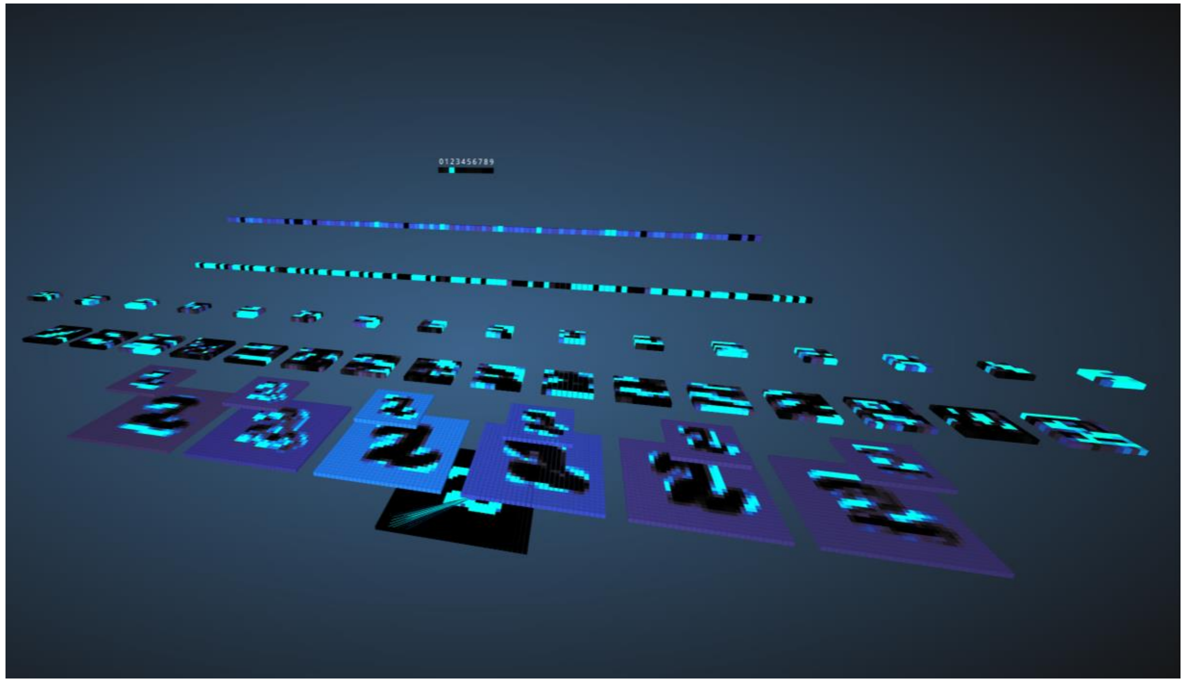
  - Contrary, using a weight sharing scheme, we have only:

$$(5 \cdot 5 + 1) \cdot 6 = 156$$



How do we use backpropagation with shared parameters?

- Every neuron in the volume will compute the gradient for its weights, but these gradients will be added up across each depth slice and **only update a single set of weights per slice**.
- **Forward pass** of one depth slice can be **computed as a convolution** of the neuron's weights with the input volume (→ Convolutional Layer, Kernel, Filter).



Visualization of the Learned Filters in the First layer

- Example filters learned by Krizhevsky *et al.* (2012). Each of the 96 filters is of size 11x11x3 and each one is shared among all neurons in one depth slice.

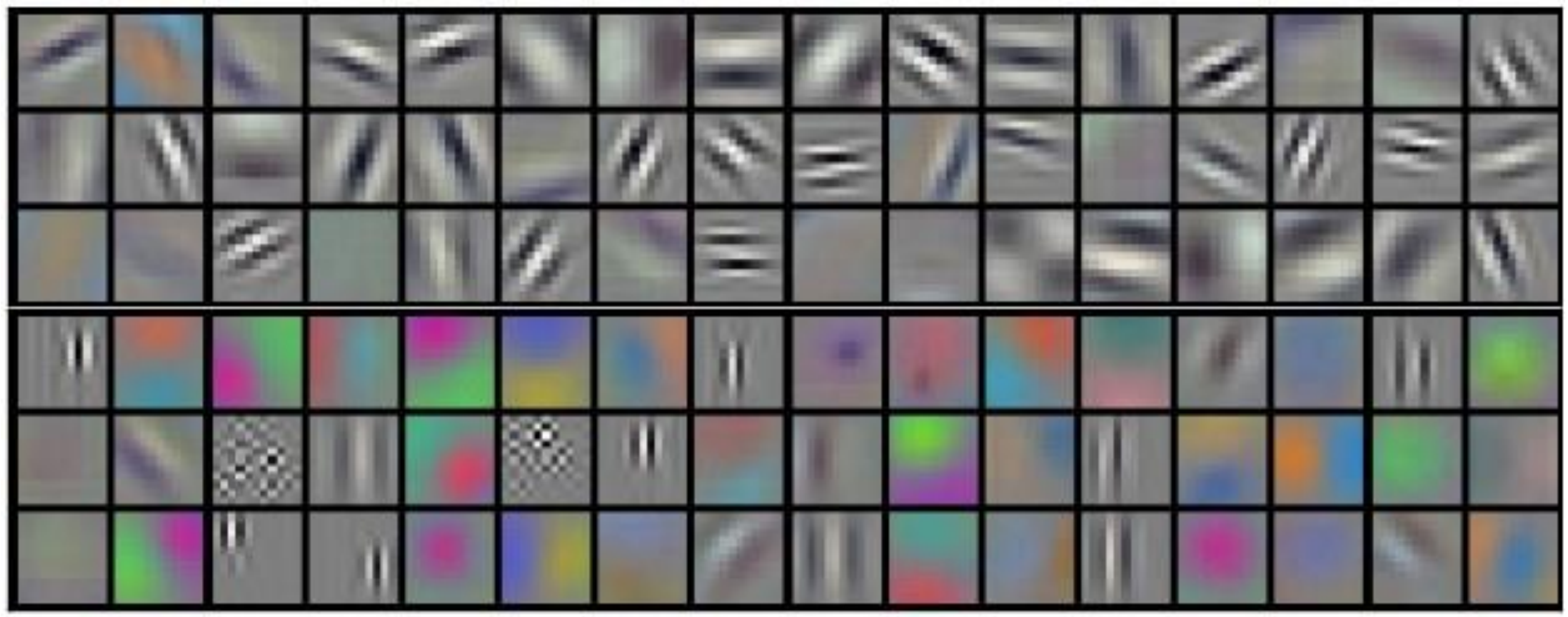


Image: Krshievsky et al., AlexNet, 2012

Summary: Convolutional Layers

- Input: Volume of size $W_1 \times H_1 \times D_1$
- Four hyperparameters:
  - Number of filters: K
  - Spatial extent (receptive field): F
  - Stride: S
  - Amount of zero-padding: P
- Output is a volume of size $W_2 \times H_2 \times D_2$ with:

$$W_2 = (W_1 - F + 2P)/S + 1$$
$$H_2 = (H_1 - F + 2P)/S + 1$$
$$D_2 = K$$

- As parameters are shared, we obtain $F \cdot F \cdot D_1$ trainable weights per filter, *i.e.*, a total of $(F \cdot F \cdot D_1) \cdot K$ weights and K bias weights.
- Each slice of the output volume is the convolution result with the d-th filter, stride S and the d-th offset.

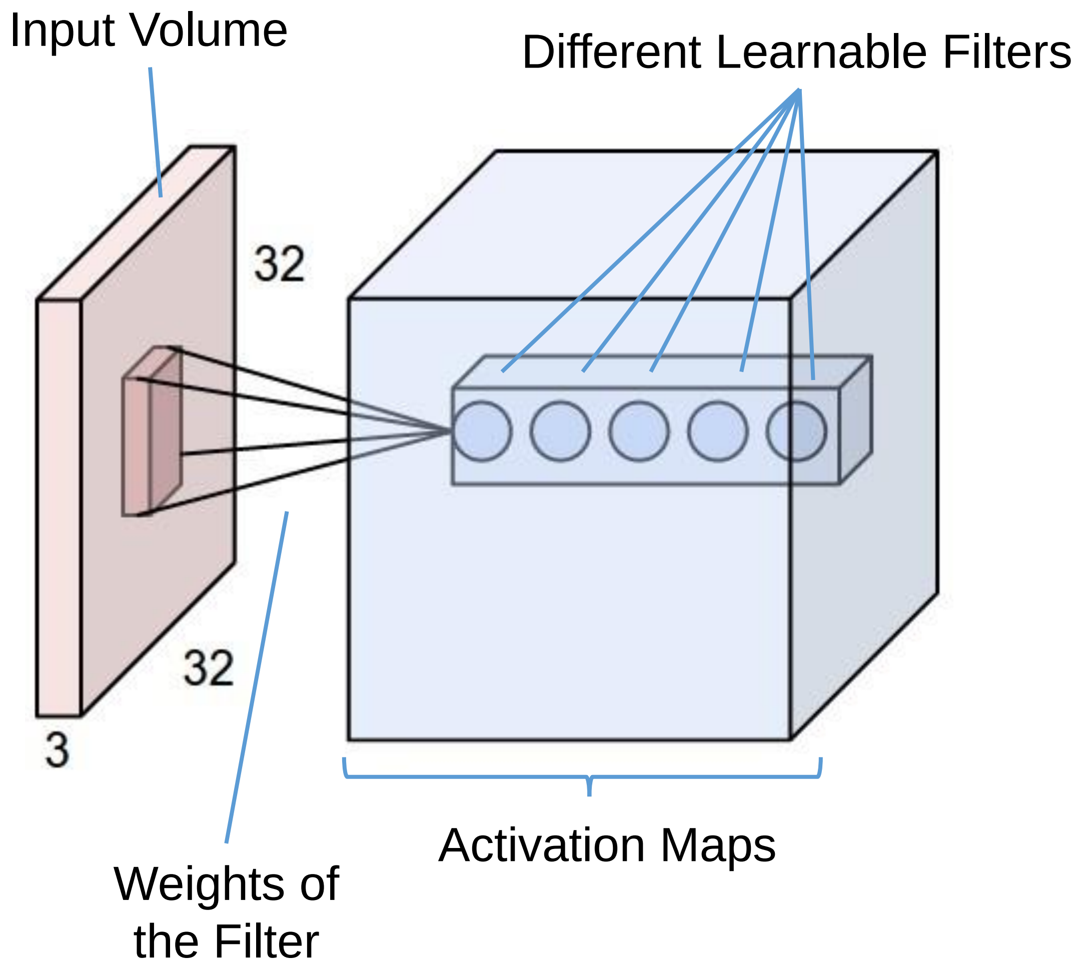


Image: http://cs231n.github.io/convolutional-networks/

Pooling Layers

Institute of Imaging and Computer Vision

RWTH AACHEN UNIVERSITY

Pooling Layers

- A typical choice after the convolutional layer is to add a pooling or subsampling layer.
- Inspired by a model of the visual cortex in mammals proposed by Hubel and Wiesel (1959): “Visual cortex consists of **simple and complex cells**. Simple cells perform feature extraction, complex cells combine (aggregate) features to a more meaningful whole.”
- Pooling is used to model this **aggregation of features** and performs a **dimensionality reduction**.
- The reduction of spatial resolution can be interpreted as raising the **translational invariance**.
- Pooled feature maps are feature maps of reduced spatial dimension, *i.e.*, they reduce the number of parameters and counteract overfitting.

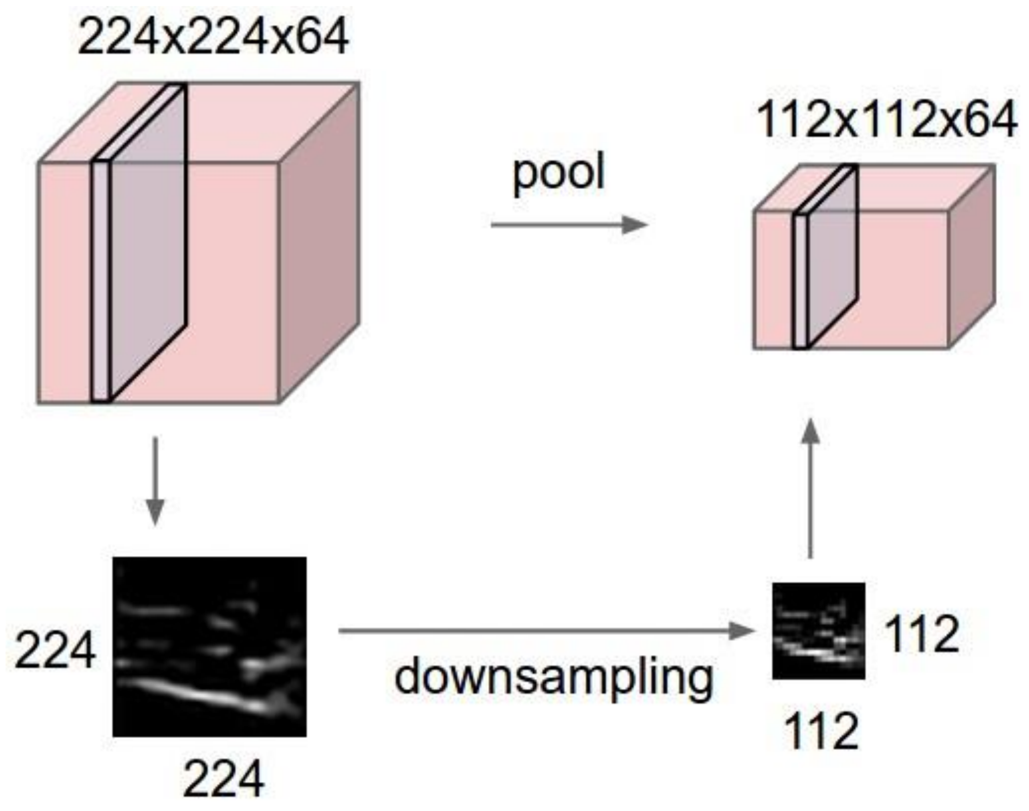
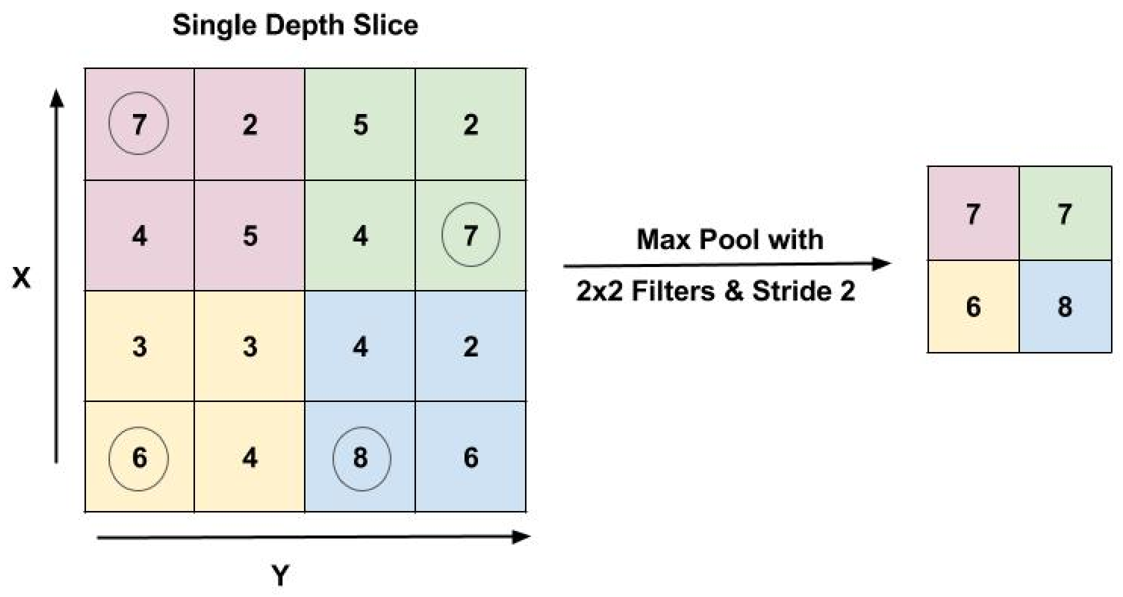


Image: http://cs231n.github.io/convolutional-networks/

Pooling Layers

- Pooling layers subdivide the input feature maps into a set of small regions (e.g., 2x2), so called **pooling neighborhoods**.
- Usually, **neighborhoods are** assumed to be **adjacent,** and elements of a pooling neighborhood are **summarized to a single value**.
- The three most common pooling methods:
  - **Average pooling**: arithmetic mean of the values within the pooling neighborhood.
  - **Max pooling**: maximum value of the pooling neighborhood (most used).
  - **L2 pooling**: square root of the sum of the squared elements in the pooling neighborhood.



Max pooling example with a pooling neighborhood of 2x2 and a stride of 2.

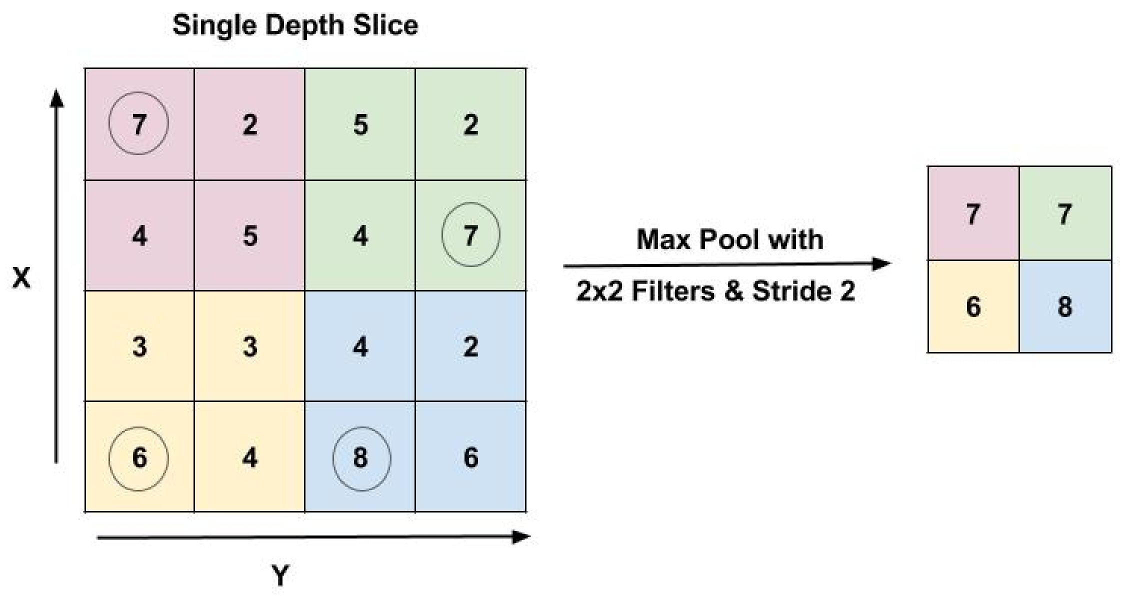
Image: http://cs231n.github.io/convolutional-networks/

Summary: Pooling Layers

- Input: Volume of size $W_1 \times H_1 \times D_1$
- Two hyperparameters:
  - Spatial extent (receptive field): F
  - Stride: S
- Output is a volume of size $W_2 \times H_2 \times D_2$ with:

$$W_2 = (W_1 - F)/S + 1$$
$$H_2 = (H_1 - F)/S + 1$$
$$D_2 = D_1$$

- Pooling layers do **not contain additional trainable parameters** and have the same depth as the input layer.
- Usually, **no padding** is used for pooling layers.
- Mostly used in practice: $F = 2, S = 2$



Max pooling example with a pooling neighborhood of 2x2 and a stride of 2.

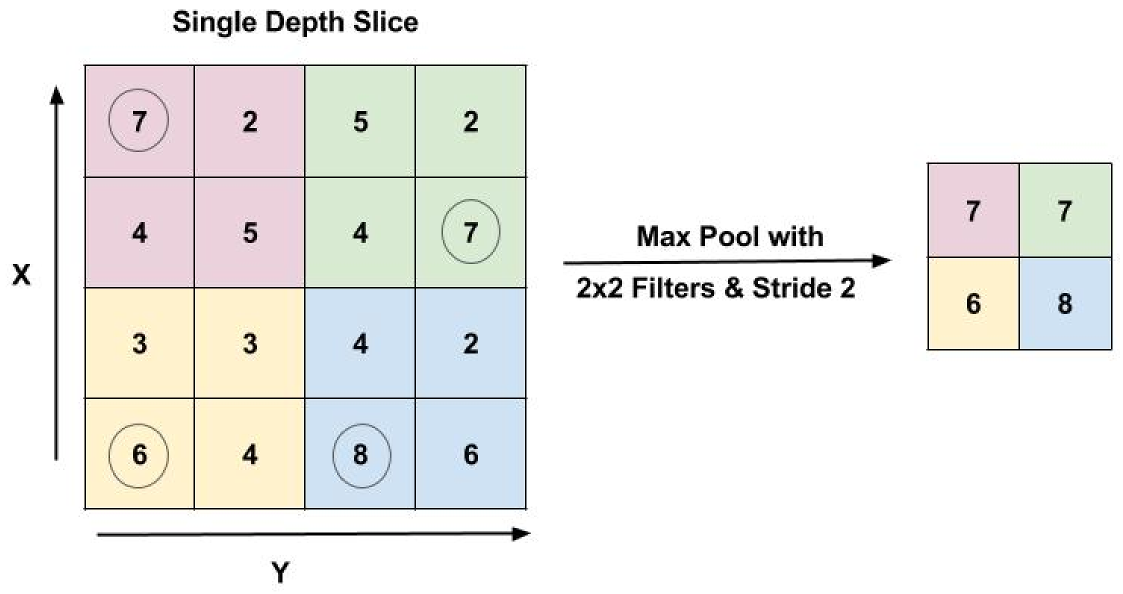
Image: http://cs231n.github.io/convolutional-networks/

Summary: Pooling Layers

- Input: Volume of size $W_1 \times H_1 \times D_1$
- Two hyperparameters:
  - Spatial extent (receptive field): F
  - Stride: S
- Output is a volume of size $W_2 \times H_2 \times D_2$ with:

$$W_2 = (W_1 - F)/S + 1$$
$$H_2 = (H_1 - F)/S + 1$$
$$D_2 = D_1$$

- Backpropagation:
  - in the backward pass, max pooling routes the gradient only to the input that yielded the maximum value.
  - Store maximum activation index during the forward pass, to have it ready for the backward pass.
- Alternative to pooling can be obtained, *e.g.*, by using only convolutional layers with strides $S > 1$.



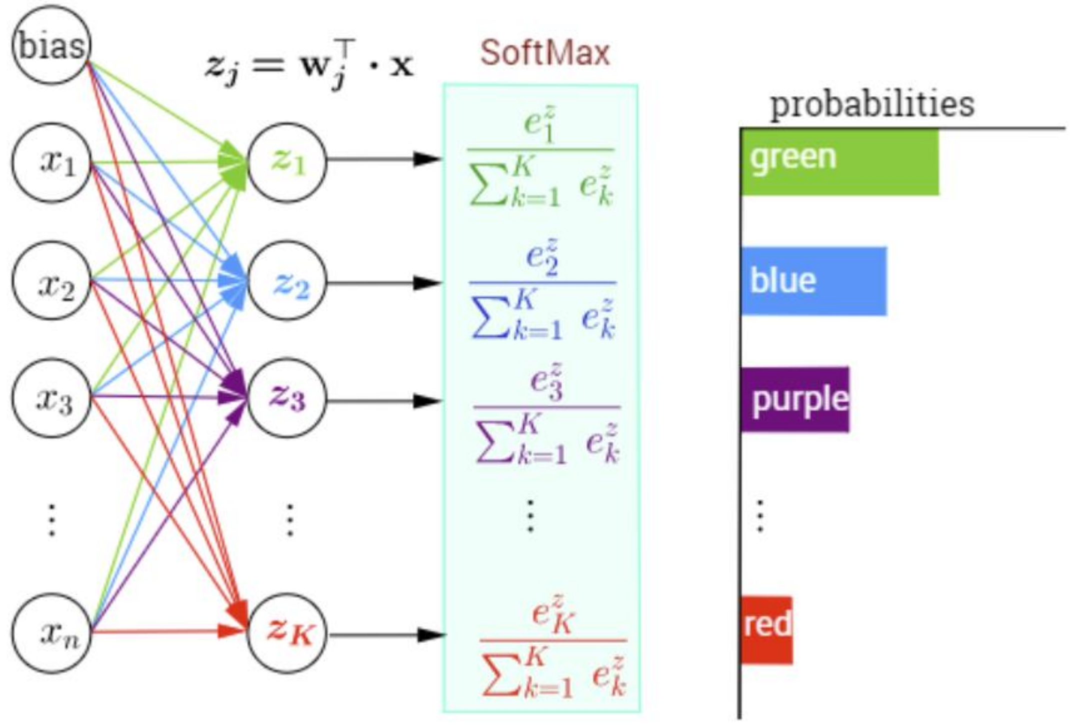
Max pooling example with a pooling neighborhood of 2x2 and a stride of 2.

Image: http://cs231n.github.io/convolutional-networks/

Fully-Connected Layers

- The final classification is usually performed using a fully-connected layer.
- This works identical to the concepts we learned in the last lecture for multilayer perceptrons!
- The last pooling layer is vectorized and then used as the input to one or more fully connected layers.
- Softmax activation can be used to convert the activations of the output neurons to a **categorical probability distribution** over the K different outcomes:

$$\sigma_j(\mathbf{z}) = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$$



bias
x_1
x_2
x_3
x_n
$z_j = \mathbf{w}_j^\top \cdot \mathbf{x}$
z_1
z_2
z_3
z_K
SoftMax
$\frac{e_1^z}{\sum_{k=1}^K e_k^z}$
$\frac{e_2^z}{\sum_{k=1}^K e_k^z}$
$\frac{e_3^z}{\sum_{k=1}^K e_k^z}$
$\frac{e_K^z}{\sum_{k=1}^K e_k^z}$
probabilities
green
blue
purple
red

Image: http://rinterested.github.io/statistics/softmax.html

Basic Architecture of Convolutional Neural Networks

- The three layer types (convolutional, pooling and fully connected) are stacked together to form a ConvNet.
- ReLU activation can be applied to each neuron of the conv layer individually (could be implemented as layer).
- Common architecture pattern: $\mathrm{INPUT} \rightarrow [[\mathrm{CONV} \rightarrow \mathrm{RELU}] \cdot N \rightarrow \mathrm{POOL?}] \cdot M \rightarrow [\mathrm{FC} \rightarrow \mathrm{RELU}] \cdot K \rightarrow \mathrm{FC}$
- Example:

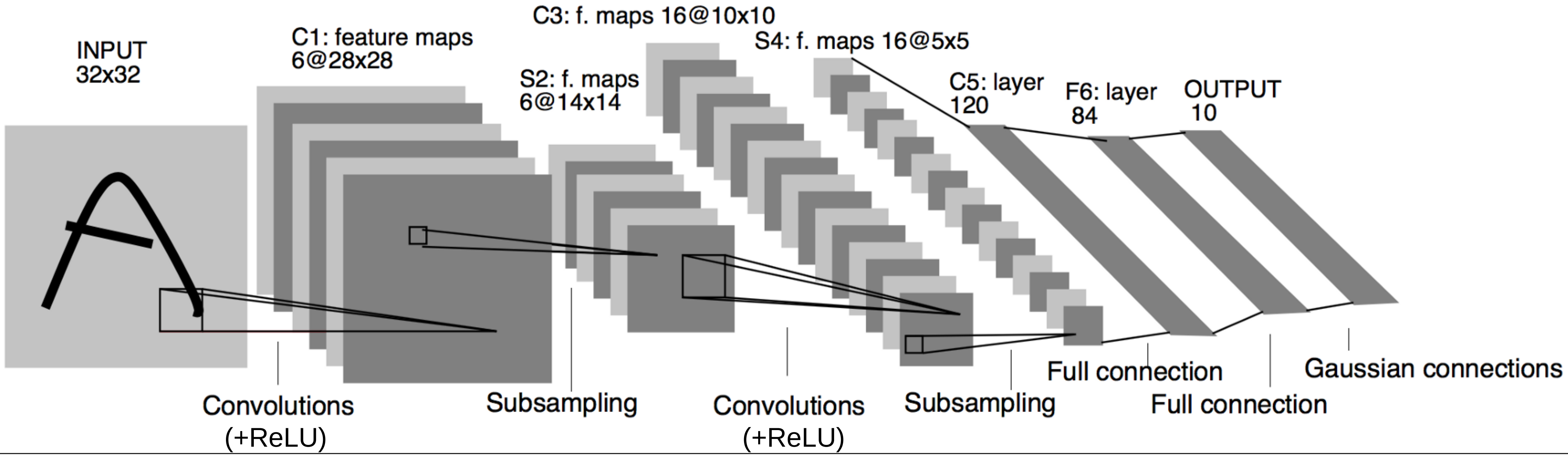


Image: https://cdn-images-1.medium.com/max/2000/1*1TI1aGBZ4dybR6__DI9dzA.png

Basic Architecture of Convolutional Neural Networks

- Prefer stack of small filters over a convolutional filter with a large receptive field.
- For instance, stacking three 3x3 convolutional layers effectively “look” at a receptive field of 7x7 in the third layer and include non-linearities between each layer.
- If this would be realized with a single 7x7 convolutional layer, we have the following disadvantages:
    - Linear function over the input rather than intermediate non-linearities of the stacked smaller filters
    - The 7x7 conv layer requires $C \times (7 \times 7 \times C) = 49C^2$ parameters, whereas the stacked layers require only $3 \times (C \times (3 \times 3 \times C)) = 27C^2$
- Thus, stacking layers with smaller receptive fields can express more powerful features with fewer parameters.

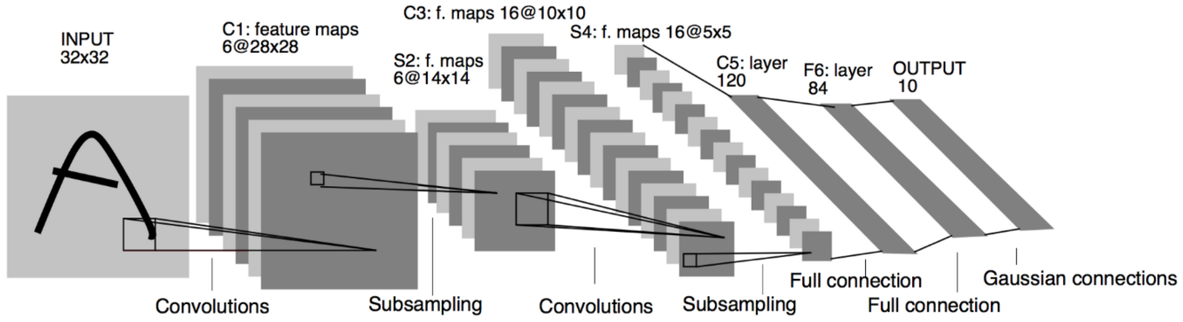


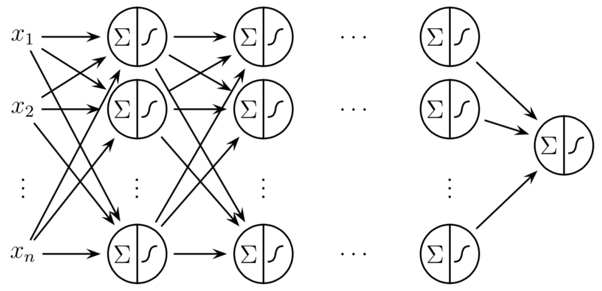
Image: https://cdn-images-1.medium.com/max/2000/1*1TI1aGBZ4dybR6__DI9dzA.png

Summary, Preview and Literature

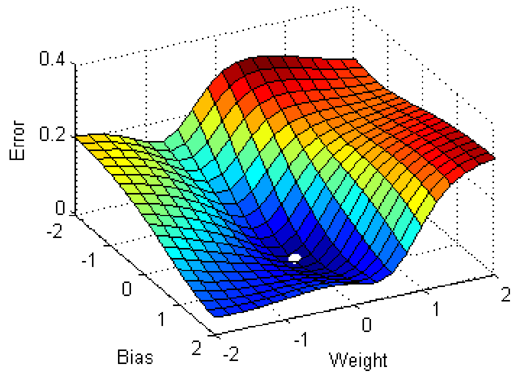
Institute of Imaging and Computer Vision

RWTH AACHEN UNIVERSITY

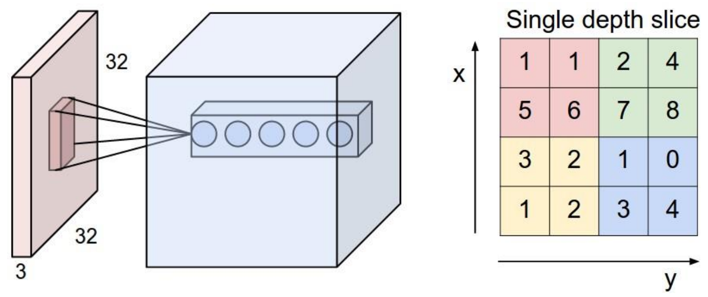
Summary: Deep Learning Crash Course



Basic ideas of artificial neural networks.

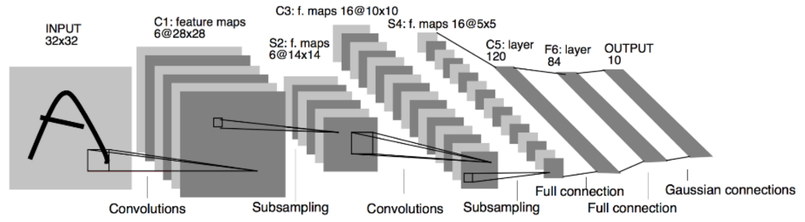


Basic optimization theory and gradient descent.



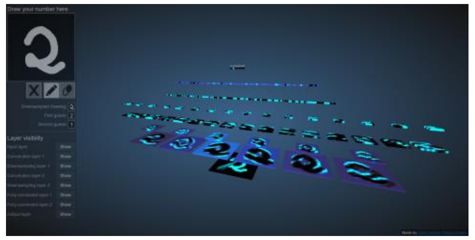
1	1	2	4
5	6	7	8
3	2	1	0
1	2	3	4

Introduction of the basic layer types of convolutional neural networks.



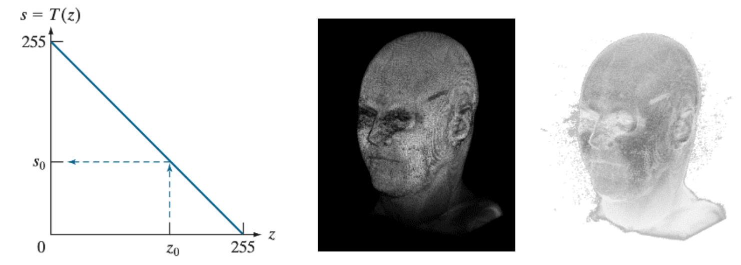
(+ReLU) (+ReLU)

Common architectures of deep neural networks.

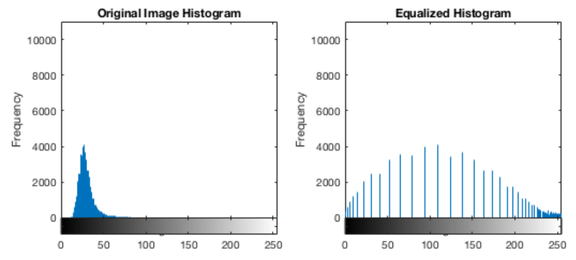


Application examples of classical and convolutional neural networks in practice.

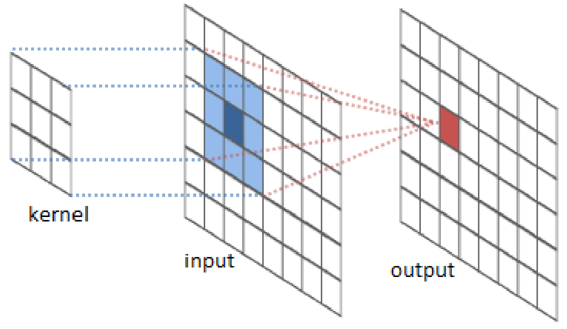
Preview: Point Operations and Spatial Filtering



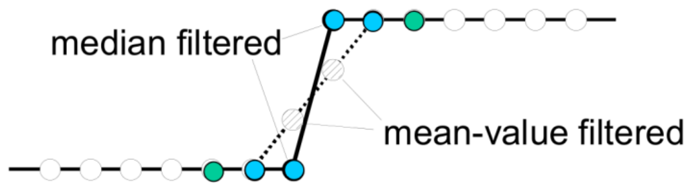
Point Operations



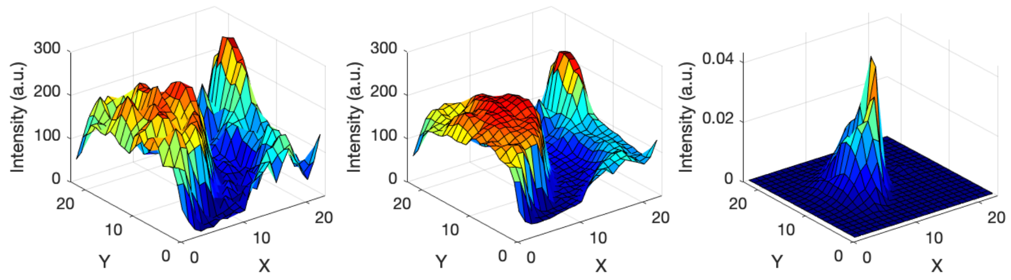
Histogram-based
Image Processing



Linear Spatial Filtering



Non-Linear Spatial Filtering



Bilateral Filter

Questions

- What are the components of a multilayer perceptron? Assume you have an architecture and pre-trained weights, how do you determine the network output given a certain input pattern?
- Can you conceptually explain the training procedure of a multilayer perceptron? Which components are actually adapted? How do you compute the prediction error made by the network for a given training sample?
- Explain the idea of gradient descent-based optimization of multidimensional functions. What is the learning rate and why is it important to choose is appropriately? Can you draw the concept of gradient descent?
- What’s the primary difference between convolutional neural networks and classical machine learning based approaches as discussed in the previous lectures?
- Can you explain the concept of convolutional layers? What are filters? How are neurons connected in the convolutional neural network setting? How does backpropagation work for shared weights?
- Which hyperparameters have to be specified for a convolutional layer? How does the number of weights depend on these hyperparameters?
- Explain the concept of pooling layers. What is the primary intention of pooling layers? Which hyperparameters have to be specified for pooling layers?
- Where do you still find fully-connected layers in convolutional neural network architectures? Which activation is commonly used for multi-category output layers of a CNN?
- Which building blocks are usually used to make up a CNN and how are they arranged to a full network?

Literature

Books

General Introduction to Image Processing:

- B. Jähne: Digital Image Processing. Springer, 2005 (more recent version available in German).
- **R.C. Gonzalez, R.E. Woods: Digital Image Processing, Global Edition, Pearson Education Limited, 4th edition, 2017.**
- W. Burger, M. J. Burge: Digitale Bildverarbeitung, Springer, 2015.
- M. Gmelch, S. Reinecke, Durchblick in Optik, Springer, 2019.
- J. Beyerer, F. Puente León, C. Frese, Machine Vision: Automated Visual Inspection: Theory, Practice and Applications, Springer, 2016

Machine Learning and Pattern Recognition:

- **C. Bishop: Pattern Recognition and Machine Learning, Springer, 2006**
- **I. Goodfellow, Y. Bengio, A. Courville: Deep Learning, MIT Press, 2016 (free version available from https://www.deeplearningbook.org/)**
- K. P. Murphy: Machine Learning: A Probabilistic Perspective, MIT Press, 2012
- T. Hastie, R. Tibshirani, J. Friedman: The Elements of Statistical Learning, Springer, 2013

Additional sources:

Talk by C. S. Perone, “Deep Learning: Convolutional Neural Networks”, 2015: https://de.slideshare.net/perone/deep-learning-convolutional-neural-networks
Blog entry by Shubham Jain, 2018, “An Overview of Regularization Techniques in Deep Learning”: https://www.analyticsvidhya.com/blog/2018/04/fundamentals-deep-learning-regularization-techniques/
Free online book by Michael Nielsen, “Neural Networks and Deep Learning”, 2018: http://neuralnetworksanddeeplearning.com

Credits:

- The lecture was partly adapted from the cs231 course held at Stanford University: http://cs231n.github.io/convolutional-networks/ .
- Thanks to the following people for their contributions to the tutorial preparation: Dennis Eschweiler, Ankit Amrutkar, Abin Jose, Martin Strauch, Priyanka Das, Ina Laube, Mathias Wien.
- The raw image on the title page was acquired by Weiyi Qian and Holger Knaut (Department of Cell Biology, NYU Langone Health, New York City, USA).